Introduction

The Definitive Guide to SQLite covers SQLite in a comprehensive fashion, giving you the knowledge and experience to use it in a wide range of situations. Whether you are a hard-core C developer, are a mobile device aficionado, or are just seeking more know-how on the best embedded and small-footprint database engine ever invented, this book it for you.

Prerequisites

This book assumes no prior knowledge of SQLite, though naturally people of all experience levels will benefit from the material. SQLite is written in C with an extensive C API and also supports many other languages such as Python, Tcl, Ruby, and Java. As a database engine, it also makes extensive use of SQL. Although the examples in this book will benefit a reader of any skill level, we don't have space to also teach those languages.

How This Book Is Organized

The book contains 11 chapters, which cover the following broad areas:

- SQLite introduction, acquisition, and installation
- Using SQL with SQLite
- The C API for SQLite
- Using languages such as Python, Tcl, Ruby, and Java with SQLite
- Mobile device development with SQLite
- Internals and new features of SQLite

There's no real impediment to jumping in to whatever area takes your fancy, though you may find that Chapters 5, 6, and 7, which deal with the C API, are best approached in order.

Obtaining the Source Code of the Examples

The source code of all the examples in the book is available from the book's catalog page on the Apress web site, at `http://apress.com/book/view/1430232250`. Look for the "Source Code" link in the "Book Resources" sidebar.

CHAPTER 1

Introducing SQLite

SQLite is an open source, embedded relational database. Originally released in 2000, it is designed to provide a convenient way for applications to manage data without the overhead that often comes with dedicated relational database management systems. SQLite has a well-deserved reputation for being highly portable, easy to use, compact, efficient, and reliable.

An Embedded Database

SQLite is an *embedded* database. Rather than running independently as a stand-alone process, it symbiotically coexists inside the application it serves—within its process space. Its code is intertwined, or *embedded*, as part of the program that hosts it. To an outside observer, it would never be apparent that such a program had a relational database management system (RDBMS) on board. The program would just do its job and manage its data somehow, making no fanfare about how it went about doing so. But inside, there is a complete, self-contained database engine at work.

One advantage of having a database server inside your program is that no network configuration or administration is required. Take a moment to think about how liberating that is: no firewalls or address resolution to worry about, and no time wasted on managing intricate permissions and privileges. Both client and server run together in the same process. This reduces overhead related to network calls, simplifies database administration, and makes it easier to deploy your application. Everything you need is compiled right into your program.

Consider the processes found in Figure 1-1. One is a Perl script, another is a standard C/C++ program, and the last is an Apache-hosted PHP script, all using SQLite. The Perl script imports the DBI::SQLite module, which in turn is linked to the SQLite C API, pulling in the SQLite library. The PHP library works similarly, as does the C++ program. Ultimately, all three processes interface with the SQLite C API. All three therefore have SQLite embedded in their process spaces. By doing so, not only does each of those processes run their own respective code, but they've also become independent database servers in and of themselves. Furthermore, even though each process represents an independent server, they can still operate on the same database file(s) , benefitting from SQLite's use of the operating system to manage synchronization and locking.

Today there is a wide variety of relational database products on the market specifically designed for embedded use—products such as Sybase SQL Anywhere, Oracle TimesTen and BerkleyDB, Pervasive PSQL, and Microsoft's Jet Engine. Some of the dominant commercial vendors have pared down their large-scale databases to create embedded variants. Examples of these include IBM's DB2 Everyplace, Oracle's Database Lite, and Microsoft's SQL Server Express. The open source databases MySQL and Firebird both offer embedded versions as well. Of all these products, only one is open source, unencumbered by licensing fees, and designed exclusively for use as an embedded database: SQLite.

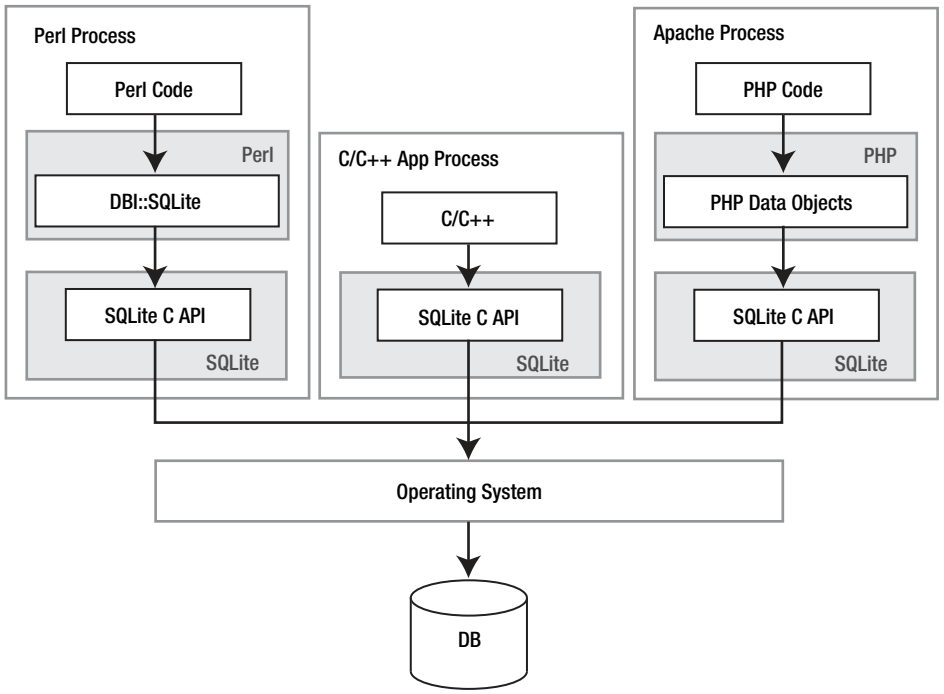


Figure 1-1. *SQLite embedded in host processes*

A Developer's Database

SQLite is quite versatile. It is a database, a programming library, and a command-line tool, as well an excellent learning tool that provides a good introduction to relational databases. There are many ways to use it—in embedded environments, websites, operating system services, scripts, and applications. For programmers, SQLite is like "data duct tape," providing an easy way to bind applications and their data. Like duct tape, there is no end to its potential uses. In a web environment, SQLite can help with managing complex session information. Rather than serializing session data into one big blob, individual pieces can be selectively written to and read from individual session databases. SQLite also serves as a good stand-in relational database for development and testing: there are no external RDBMSs or networking to configure or usernames and passwords to hinder the programmer's focus. SQLite can also serve as a cache, hold configuration data, or, by leveraging its binary compatibility across platforms, even work as an application file format.

Besides being just a storage receptacle, SQLite can serve as a purely functional tool for general data processing. Depending on size and complexity, it may be easier to represent some application data structures as a table or tables in an in-memory database. With so many developers, analysts, and others familiar with relational databases and SQL, you can benefit from "assumed knowledge"—operating on the data relationally by using SQLite to do the heavy lifting rather than having to write your own algorithms to manipulate and sort data structures. If you are a programmer, imagine how much code it would take to implement the following SQL statement in your program:

```
SELECT x, STDDEV(w)
FROM table
GROUP BY x
HAVING x > MIN(z) OR x < MAX(y)
ORDER BY y DESC
LIMIT 10 OFFSET 3;
```

If you are already familiar with SQL, imagine coding the equivalent of a subquery, compound query, `GROUP BY` clause, or multiway join in your favorite (or not so favorite) programming language. SQLite embeds all of this functionality into your application with minimal cost. With a database engine integrated directly into your code, you can begin to think of SQL as an offload engine in which to implement complex sorting algorithms in your program. This approach becomes more appealing as the size of your data set grows or as your algorithms become more complex. What's more, SQLite can be configured to use a fixed amount of RAM and then offload data to disk if it exceeds the specified limit. This is even harder to do if you write your own algorithms. With SQLite, this feature is available with a simple call to a single SQL command.

SQLite is also a great learning tool for programmers—a cornucopia for studying computer science topics. From parser generators to tokenizers, virtual machines, B-tree algorithms, caching, program architecture, and more, it is a fantastic vehicle for exploring many well-established computer science concepts. Its modularity, small size, and simplicity make it easy to present each topic as an isolated case study that any individual could easily follow.

An Administrator's Database

SQLite is not just a programmer's database. It is a useful tool for system administrators as well. It is small, compact, and elegant like finely honed versatile utilities such as `find`, `rsync`, and `grep`. SQLite has a command-line utility that can be used from the shell or command line and within shell scripts. However, it works even better with a large variety of scripting languages such as Perl, Python, TCL, and Ruby. Together the two can help with pretty much any task you can imagine, such as aggregating log file data, monitoring disk quotas, or performing bandwidth accounting in shared networks. Furthermore, since SQLite databases are ordinary disk files, they are easy to work with, transport, and back up.

SQLite is a convenient learning tool for administrators looking to learn more about relational databases. It is an ideal beginner's database with which to learn about relational concepts and practice their implementation. It can be installed quickly and easily on any platform you're likely to encounter, and its database files share freely between them without the need for conversion. It is full featured but not daunting. And SQLite—both the program and the database—can be carried around on a USB stick or memory chip.

SQLite History

SQLite was conceived on a battleship…well, sort of. SQLite's author, D. Richard Hipp, was working for General Dynamics on a program for the U.S. Navy developing software for use on board guided missile destroyers. The program originally ran on Hewlett-Packard Unix (HP-UX) and used an Informix database as the back end. For their particular application, Informix was somewhat overkill. For an experienced database administrator (DBA) at the time, it could take almost an entire day to install or upgrade. To the uninitiated application programmer, it might take forever. What was really needed was a self-contained database that was easy to use and that could travel with the program and run anywhere regardless of what other software was or wasn't installed on the system.

In January 2000, Hipp and a colleague discussed the idea of creating a simple embedded SQL database that would use the GNU DBM hash library (gdbm) as a back end, one that would require no installation or administrative support whatsoever. Later, when some free time opened up, Hipp started work on the project, and in August 2000, SQLite 1.0 was released.

As planned, SQLite 1.0 used `gdbm` as its storage manager. However, Hipp soon replaced it with his own B-tree implementation that supported transactions and stored records in key order. With the first major upgrade in hand, SQLite began a steady evolution, growing in both features and users. By mid-2001, many projects—both open source and commercial alike—started to use it. In the years that followed, other members of the open source community started to write SQLite extensions for their favorite scripting languages and libraries. One by one, new extensions for popular languages and APIs such as Open Database Connectivity (ODBC), Perl, Python, Ruby, Java, and others fell into place and testified to SQLite's wide application and utility.

SQLite began a major upgrade from version 2 to 3 in 2004. Its primary goal was enhanced internationalization supporting UTF-8 and UTF-16 text as well as user-defined text-collating sequences. Although version 3.0 was originally slated for release in summer 2005, America Online provided the necessary funding to see that it was completed by July 2004. Besides internationalization, version 3 brought many other new features such as a revamped C API, a more compact format for database files (a 25 percent size reduction), manifest typing, binary large object (BLOB) support, 64-bit ROWIDs, autovacuum, and improved concurrency. Even with many new features, the overall library footprint was still less than 240KB, at a time when most home PCs began measuring their memory in gigabytes! Another improvement in version 3 was a good code cleanup—revisiting, refactoring and rewriting, or otherwise throwing out the detritus accumulated in the 2.*x* series.

SQLite continues to grow feature-wise while still remaining true to its initial design goals: simplicity, flexibility, compactness, speed, and overall ease of use. At the time of this book's latest edition, SQLite has leapt ahead to include such advanced features as recursive triggers, distribution histograms to help the optimizer produce even faster queries, and asynchronous I/O on operating systems capable of supporting such workloads. What's next after that? Well, it all depends. Perhaps you or your company will sponsor the next big feature that makes this super-efficient database even better.

Who Uses SQLite

Today, SQLite is used in a wide variety of software and products. It is used in Apple's Mac OS X operating system, Safari web browser, Mail.app email program, and RSS manager, as well as Apple's Aperture photography software. Perhaps Apple's biggest use for SQLite has come in the iPhone age. You'll find many apps on the iPhone, such as the Contacts database, Notes, and more, all rely on SQLite. This reliance on SQLite also extends to the iPad. We'll return to this topic in Chapter 9, where we'll discuss working with SQLite on Apple's mobile platforms in detail.

SQLite can be found in Sun's Solaris operating environment, specifically the database backing the Service Management Facility that debuted with Solaris 10, a core component of its predictive self-healing technology. SQLite is in the Mozilla Project's mozStorage C++/JavaScript API layer, which will be the backbone of personal information storage for Firefox, Thunderbird, and Sunbird. SQLite has been added as part of the PHP 5 standard library. It also ships as part of Trolltech's cross-platform Qt C++ application framework, which is the foundation of the popular KDE window manager, and many other software applications. SQLite is especially popular in embedded platforms. Much of Richard Hipp's SQLite-related business has been porting SQLite to various proprietary embedded platforms. Symbian uses SQLite to provide SQL support in the native Symbian OS platform. Google has made extensive use of SQLite in the Android mobile phone operating system and user-space applications. SQLite is so pervasive within Android that Chapter 10 is dedicated to showing you all about its use on Android devices. SQLite is also included in commercial development products for cell phone applications.

Although it is rarely advertised, SQLite is also used in a variety of consumer products, as some tech-savvy consumers have discovered in the course of poking around under the hood. Examples include the D-Link Media Lounge, the Slim Devices Squeezebox music player, and the Philips GoGear personal music player. Some clever consumers have even found a SQLite database embedded in the *Complete New Yorker* DVD set—a digital library of every issue of the *New Yorker* magazine—apparently used by its accompanying search software.

You can find SQLite as an alternative back-end storage facility for a wide array of open source projects such as Yum (the package manager for Fedora Core), Movable Type, DSPAM, and Edgewall Software's excellent Trac SCM and project management system; possibly most famously, it's used as the built-in database for Firefox, the web browser from Mozilla. Even parts of SQLite's core utilities can be found in other open source projects. One such example is its Lemon parser generator, which the lighttpd web server project uses for generating the parser code for reading its configuration file. Indeed, there seems to be such a variety of uses for SQLite that Google took notice and awarded Richard Hipp with "Best Integrator" at O'Reilly's 2005 Open Source Convention—long before Google entered the mobile space with Android. See also `www.sqlite.org/famous.html` for other ideas of important SQLite users.

Architecture

SQLite has an elegant, modular architecture that takes some unique approaches to relational database management. It consists of eight separate modules grouped within three major subsystems (as shown in Figure 1-2). These modules divide query processing into discrete tasks that work like an assembly line. The top of the stack compiles the query, the middle executes it, and the bottom handles storage and interfacing with the operating system.

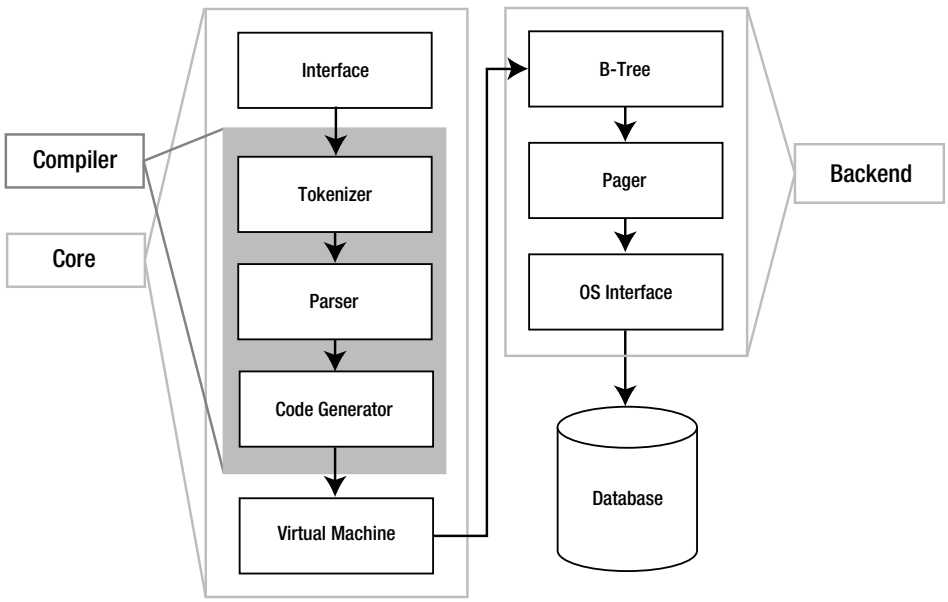


Figure 1-2. *SQLite's architecture*

The Interface

The interface is the top of the stack and consists of the SQLite C API. It is the means through which programs, scripting languages, and libraries alike interact with SQLite. Literally, this is where you as developer, administrator, student, or mad scientist talk to SQLite.

The Compiler

The compilation process starts with the tokenizer and parser. They work together to take a Structured Query Language (SQL) statement in text form, validate its syntax, and then convert it to a hierarchical data structure that the lower layers can more easily manipulate. SQLite's tokenizer is hand-coded. Its parser is generated by SQLite's custom parser generator, which is called Lemon. The Lemon parser generator is designed for high performance and takes special precautions to guard against memory leaks. Once the statement has been broken into tokens, evaluated, and recast in the form of a parse tree, the parser passes the tree down to the code generator.

The code generator translates the parse tree into a kind of assembly language specific to SQLite. This assembly language consists of instructions that are executable by its virtual machine. The code generator's sole job is to convert the parse tree into a complete mini-program written in this assembly language and to hand it off to the virtual machine for processing.

The Virtual Machine

At the center of the stack is the virtual machine, also called the *virtual database engine* (VDBE). The VDBE is a register-based virtual machine that works on byte code, making it independent of the underlying operating system, CPU, or system architecture. The VDBE's byte code (or virtual machine language) consists of more than 100 possible tasks known as *opcodes*, which are all centered on database operations. The VDBE is designed specifically for data processing. Every instruction in its instruction set either accomplishes a specific database operation (such as opening a cursor on a table, making a record, extracting a column, or beginning a transaction) or performs manipulations to prepare for such an operation. Altogether and in the right order, the VDBE's instruction set can satisfy any SQL command, however complex. Every SQL statement in SQLite—from selecting and updating rows to creating tables, views, and indexes—is first compiled into this virtual machine language, forming a stand-alone instruction set that defines how to perform the given command. For example, take the following statement:

```
SELECT name FROM episodes LIMIT 10;
```

This compiles into the VDBE program shown in Listing 1-1.

Listing 1-1. *VDBE Assembly*

```
addr        opcode      p1          p2          p3          p4          p5          comment
0           Trace       0           0           0                       00
1           Integer     10          1           0                       00
2           Goto        0           11          0                       00
3           OpenRead    0           2           0           3           00
4           Rewind      0           9           0                       00
5           Column      0           2           2                       00
6           ResultRow   2           1           0                       00
7           IfZero      1           9           -1                      00
8           Next        0           5           0                       01
9           Close       0           0           0                       00
10          Halt        0           0           0                       00
11          Transactio  0           0           0                       00
12          VerifyCook  0           4           0                       00
13          TableLock   0           2           0           episodes    00
14          Goto        0           3           0                       00
```

The program consists of 15 instructions. These instructions, performed in this particular order with the given operands, will return the `name` field of the first ten records in the `episodes` table (which is part of the example database included with this book).

In many ways, the VDBE is the heart of SQLite. All of the modules before it work to create a VDBE program, while all modules after it exist to execute that program, one instruction at a time.

The Back End

The back end consists of the B-tree, page cache, and OS interface. The B-tree and page cache (pager) work together as information brokers. Their currency is database pages, which are uniformly sized blocks of data that, like shipping containers, are made for transportation. Inside the pages are the goods: more interesting bits of information such as records and columns and index entries. Neither the B-tree nor the pager has any knowledge of the contents. They only move and order pages; they don't care what's inside.

The B-tree's job is order. It maintains many complex and intricate relationships between pages, which keeps everything connected and easy to locate. It organizes pages into tree-like structures (which is the reason for the name), which are highly optimized for searching. The pager serves the B-tree, feeding it pages. Its job is transportation and efficiency. The pager transfers pages to and from disk at the B-tree's behest. Disk operations are still some of the slowest things a computer has to do, even with today's solid-state disks. Therefore, the pager tries to speed this up by keeping frequently used pages cached in memory and thus minimizes the number of times it has to deal directly with the hard drive. It uses special techniques to predict which pages will be needed in the future and thus anticipate the needs of the B-tree, keeping pages flying as fast as possible. Also in the pager's job description are transaction management, database locking, and crash recovery. Many of these jobs are mediated by the OS interface.

Things such as file locking are often implemented differently in different operating systems. The OS interface provides an abstraction layer that hides these differences from the other SQLite modules. The end result is that the other modules see a single consistent interface with which to do things like file locking. So, the pager, for example, doesn't have to worry about doing file locking one way on Windows and doing it another way on different operating systems such as Unix. It lets the OS interface worry about this. It just says to the OS interface, "Lock this file," and the OS interface figures out how to do that

based on the operating system on which it happens to be running. Not only does the OS interface keep code simple and tidy in the other modules, but it also keeps the messy issues cleanly organized and at arm's length in one place. This makes it easier to port SQLite to different operating systems—all of the OS issues that must be addressed are clearly identified and documented in the OS interface's API.

Utilities and Test Code

Miscellaneous utilities and common services such as memory allocation, string comparison, and Unicode conversion routines are kept in the utilities module. This is basically a catchall module for services that multiple modules need to use or share. The testing module contains a myriad of regression tests designed to examine every little corner of the database code. This module is one of the reasons SQLite is so reliable: it performs a lot of regression testing and makes those tests available for anyone to run and improve.

SQLite's Features and Philosophy

SQLite offers a surprisingly comprehensive range of features and capabilities despite its small size. It supports a large subset of the ANSI SQL92 standard for SQL features (transactions, views, check constraints, foreign keys, correlated subqueries, compound queries, and more) along with many other features found in relational databases, such as triggers, indexes, autoincrement columns, and `LIMIT/OFFSET` features. It also has many rare or unique features, such as in-memory databases, dynamic typing, and conflict resolution—otherwise referred to as `merge` or `upsert` in other RDBMSs—which will be explained in a moment.

As mentioned earlier in this chapter, SQLite has a number of governing principles or characteristics that serve to more or less define its philosophy and implementation. I'll expand on these issues next.

Zero Configuration

From its initial conception, SQLite has been designed so that it can be incorporated and used without the need of a DBA. Configuring and administering SQLite is as simple as it gets. SQLite contains just enough features to fit in a single programmer's brain, and like its library, it requires as small a footprint in the gray matter as it does in RAM.

Portability

SQLite was designed specifically with portability in mind. It compiles and runs on Windows, Linux, BSD, Mac OS X, and commercial Unix systems such as Solaris, HP-UX, and AIX, as well as many embedded platforms such as QNX, VxWorks, Symbian, Palm OS, and Windows CE. It works seamlessly on 32- and 64-bit architectures with both big- and little-endian byte orders. Portability doesn't stop with the software either: SQLite's database files are as portable as its code. The database file format is binary compatible across all supported operating systems, hardware architectures, and byte orders. You can create a SQLite database on a Linux workstation and use it on a Mac or Windows machine, an iPhone, or other device, without any conversion or modification. Furthermore, SQLite databases can hold up to 2 terabytes of data (limited only by the operating system's maximum file size) and natively support both UTF-8 and UTF-16 encoding.

Compactness

SQLite was designed to be lightweight and self-contained; one header file, one library, and you're relational—no external database server required. Everything packs into less than half a megabyte, which is smaller than many of the web pages you'll visit on any given day.

SQLite databases are ordinary operating system files. Regardless of your system, all objects in your SQLite database—tables, triggers, schema, indexes, and views—are contained in a single operating system file. SQLite uses variable-length records wherever possible, allocating only the minimum amount of data needed to hold each field. A 2-byte field sitting in a `varchar(100)` column takes up only 3 bytes of space, not 100 (the extra byte is used to record its type information).

Simplicity

As a programming library, SQLite's API is one of the simplest and easiest to use. The API is both well documented and intuitive. It is designed to help you customize SQLite in many ways, such as implementing your own custom SQL functions in C. The open source community also has a created a vast number of language and library interfaces with which to use SQLite. There are extensions for Perl, Python, Ruby, Tcl/Tk, Java, PHP, Visual Basic, ODBC, Delphi, C#, VB .NET, Smalltalk, Ada, Objective C, Eiffel, Rexx, Lisp, Scheme, Lua, Pike, Objective Camel, Qt, WxWindows, REALBASIC, and others. You can find an exhaustive list on the SQLite wiki: `www.sqlite.org/cvstrac/wiki?p=SqliteWrappers`.

SQLite's modular design includes many innovative ideas that enable it to be full featured and extensible while at the same time retaining a great degree of simplicity throughout its code base. Each module is a specialized, independent system that performs a specific task. This modularity makes it much easier to develop each system independently and to debug queries as they pass from one module to the next—from compilation and planning to execution and materialization. The end result is that there is a crisp, well-defined separation between the front end (SQL compiler) and back end (storage system), allowing the two to be coded independently of each other. This design makes it easier to add new features to the database engine, is faster to debug, and results in better overall reliability.

Flexibility

Several factors work together to make SQLite a very flexible database. As an embedded database, it offers the best of both worlds: the power and flexibility of a relational database front end, with the simplicity and compactness of a B-tree back end. With it, there are no large database servers to configure, no networking or connectivity problems to worry about, no platform limitations, and no license fees or royalties to pay. Rather, you get simple SQL support dropped right into your application.

Liberal Licensing

All of SQLite's code is in the public domain. There is no license. No claim of copyright is made on any part of the core source code. All contributors to this code are required to sign affidavits specifically disavowing any copyright interest in contributed code. Thus, there are no legal restrictions on how you may use the source code in any form. You can modify, incorporate, distribute, sell, and use the code for any purpose—commercial or otherwise—without any royalty fees or restrictions.

Reliability

The SQLite source code is more than just free; it also happens to be well written. SQLite's code base consists of about 70,000 lines of standard ANSI C that are clean, modular, and well commented. The code base is designed to be approachable, easy to understand, easy to customize, and generally very accessible. It is easily within the ability of a competent C programmer to follow any part of SQLite or the whole of it with sufficient time and study.

Additionally, SQLite's code offers a full-featured API specifically for customizing and extending SQLite through the addition of user-defined functions, aggregates, and collating sequences along with support for operational security.

While SQLite's modular design significantly contributes to its overall reliability, its source code is also well tested. Whereas the core software (library and utilities) consists of about 70,000 lines of code, the distribution includes an extensive test suite consisting of more than 45 *million* lines of test code, which as of July 2009 covers 100 percent of the core code. Ask any developer how hard it is to be that comprehensive when testing a nontrivial amount of code, and you can see why people have rock-solid confidence in the reliability of SQLite.

Convenience

SQLite also has a number of unique features that provide a great degree of convenience. These include dynamic typing, conflict resolution, and the ability to "attach" multiple databases to a single session.

SQLite's dynamic typing is somewhat akin to that found in scripting languages (e.g., *duck typing* in Ruby). Specifically, the type of a variable is determined by its value, not by a declaration as employed in statically typed languages. Most database systems restrict a field's value to the type declared in its respective column. For example, each field in an integer column can hold only integers or possibly NULL. In SQLite, while a column can have a declared type, fields are free to deviate from them, just as a variable in a scripting language can be reassigned a value with a different type. This can be especially helpful for prototyping, since SQLite does not force you to explicitly change a column's type. You need only change how your program stores information in that column rather than continually having to update the schema and reload your data.

Conflict resolution is another great feature. It can make writing SQL, as easy as it is, even easier. This feature is built into many SQL operations and can be made to perform what can be called *lazy updates*. Say you have a record you need to insert, but you are not sure whether one just like it already exists in the database. Rather than write a `SELECT` statement to look for a match and then recast your `INSERT` to an `UPDATE` if it does, conflict resolution lets you say to SQLite, "Here, try to insert this record, and if you find one with the same key, just update it with these values instead." Now you've gone from having to code three different SQL statements to cover all the bases (i.e., `SELECT`, `INSERT`, and possibly `UPDATE`) to just one: `INSERT OR REPLACE (...)`. Other relational database systems mimic this aspect of conflict resolution with `UPSERT` or `MERGE` statements, but SQLite goes one step further. You can build conflict resolution into the table definition itself and dispense with the need to ever specify it again on future `INSERT` statements. In fact, you can dispense with ever having to write `UPDATE` statements to this table again—just write `INSERT` statements and let SQLite do the dirty work of figuring out what to do using the conflict resolution rules defined in the schema.

Finally, SQLite lets you "attach" external databases to your current session. Say you are connected to one database (`foo.db`) and need to work on another (`bar.db`). Rather than opening a separate connection and juggling multiple connection handles in your code, you can simply attach the database of interest to your current connection with a single SQL command:

```
ATTACH database bar.db as bar;
```

Download from Wow! eBook <www.wowebook.com>

All of the tables in `bar.db` are now accessible as if they existed in `foo.db`. You can detach it just as easily when you're done. This makes all sorts of things like copying tables between databases very easy.

Performance and Limitations

SQLite is a speedy database. But the words *speedy*, *fast*, *peppy*, or *quick* are rather subjective terms. To be perfectly honest, there are things SQLite can do faster than other databases, and there are things that it cannot. Suffice it to say, within the parameters for which it has been designed, SQLite can be said to be consistently fast and efficient across the board. SQLite uses B-trees for indexes and B+-trees for tables, the same as most other database systems. For searching a single table, it is as fast if not faster than any other database on average. Simple `SELECT`, `INSERT`, and `UPDATE` statements are extremely quick—virtually at the speed of RAM (for in-memory databases) or disk. Here SQLite is often faster than other databases, because it has less overhead to deal with in starting a transaction or generating a query plan and because it doesn't incur the overhead of making a network calls to the server or negotiating authentication and privileges. Its simplicity here makes it fast.

As queries become larger and more complex, however, query time overshadows the network call or transaction overhead, and the game goes to the database with the best optimizer. This is where larger, more sophisticated databases begin to shine. While SQLite can certainly do complex queries, it does not have a sophisticated optimizer or query planner. You can always trust SQLite to give you the result, but what it won't do is try to determine optimal paths by computing millions of alternative query plans and selecting the fastest candidate, as you might expect from Oracle or PostgreSQL. Thus, if you are running complex queries on large data sets, odds are that SQLite is not going to be as fast as databases with sophisticated optimizers.

So, there are situations where SQLite is not as fast as larger databases. But many if not all of these conditions are to be expected. SQLite is an embedded database designed for small to medium-sized applications. These limitations are in line with its intended purpose. Many new users make the mistake of assuming that they can use SQLite as a drop-in replacement for larger relational databases. Sometimes you can; sometimes you can't. It all depends on what you are trying to do.

In general, there are two major variables that define SQLite's main limitations:

- **Concurrency.** SQLite has coarse-grained locking, which allows multiple readers but only one writer at a time. Writers exclusively lock the database during writes, and no one else has access during that time. SQLite does take steps to minimize the amount of time in which exclusive locks are held. Generally, locks in SQLite are kept for only a few milliseconds. But as a general rule of thumb, if your application has high write concurrency (many connections competing to write to the same database) and it is time critical, you probably need another database. It is really a matter of testing your application to know what kind of performance you can get. We have seen SQLite handle more than 500 transactions per second for 100 concurrent connections in simple web applications. But your transactions may differ in the number of records being modified or the number and complexity of the queries involved. Acceptable concurrency all depends on your particular application and can be determined empirically only by direct testing. In general, this is true with any database: you don't know what kind of performance your application will get until you do real-world tests.

- **Networking.** Although SQLite databases can be shared over network file systems, the latency associated with such file systems can cause performance to suffer. Worse, bugs in network file system implementations can also make opening and modifying remote files—SQLite or otherwise—error prone. If the file system's locking does not work properly, two clients may be allowed to simultaneously modify the same database file, which will almost certainly result in database corruption. It is not that SQLite is incapable of working over a network file system because of anything in its implementation. Rather, SQLite is at the mercy of the underlying file system and wire protocol, and those technologies are not always perfect. For instance, many versions of NFS have a flawed `fcntl()` implementation, meaning that locking does not behave as intended. Newer NFS versions, such are Solaris NFS v4, work just fine and reliably implement the requisite locking mechanisms needed by SQLite. However, the SQLite developers have neither the time nor the resources to certify that any given network file system works flawlessly in all cases.

Again, most of these limitations are intentional, resulting from SQLite's design. Supporting high write concurrency, for example, brings with it great deal of complexity, and this runs counter to SQLite's simplicity in design. Similarly, being an embedded database, SQLite intentionally does not support networking. This should come as no surprise. In short, what SQLite can't do is a direct result of what it can. It was designed to operate as a modular, simple, compact, and easy-to-use embedded relational database whose code base is within the reach of the programmers using it. And in many respects, it can do what many other databases cannot, such as run in embedded environments where actual *power consumption* is a limiting factor.

While SQLite's SQL implementation is quite good, there are some things it currently does not implement. These are as follows:

- **Complete trigger support.** SQLite supports almost all the standard features for triggers, including recursive triggers and `INSTEAD OF` triggers. However, for all trigger types, SQLite currently requires `FOR EACH ROW` behavior, where the triggered action is evaluated for every row affected by the triggering query. ANSI SQL 92 also describes a `FOR EACH STATEMENT` trigger style, which is not currently supported.

- **Complete ALTER TABLE support.** Only the `RENAME TABLE` and `ADD COLUMN` variants of the `ALTER TABLE` command are supported. Other kinds of `ALTER TABLE` operations such as `DROP COLUMN`, `ALTER COLUMN`, and `ADD CONSTRAINT` are not implemented.

- **RIGHT and FULL OUTER JOIN.** `LEFT OUTER JOIN` is implemented, but `RIGHT OUTER JOIN` and `FULL OUTER JOIN` are not. Every `RIGHT OUTER JOIN` has a provably semantically identical `LEFT OUTER JOIN`, and vice versa. Any `RIGHT OUT JOIN` can be implemented as a `LEFT OUTER JOIN` by simply reversing the order of the tables and modifying the join constraint. A `FULL OUTER JOIN` can be implemented as a combination of two `LEFT OUTER JOIN` statements, a `UNION`, and appropriate `NULL` filtering in the `WHERE` clause.

- **Updatable views.** Views in SQLite are read-only. You may not execute a `DELETE`, `INSERT`, or `UPDATE` statement on a view. But you can create a trigger that fires on an attempt to `DELETE`, `INSERT`, or `UPDATE` a view and do what you need in the body of the trigger.

- **Windowing functions.** One of the new feature sets specified in ANSI SQL 99 are the windowing functions. These provide post-processing analytic functions for results, such a ranking, rolling averages, lag and lead calculation, and so on. SQLite currently only targets ANSI SQL 92 compliance, so it doesn't support windowing functions like `RANK()`, `ROW_NUMBER()`, and so on.
- **GRANT and REVOKE.** Since SQLite reads and writes an ordinary disk file, the only access permissions that can be applied are the normal file access permissions of the underlying operating system. `GRANT` and `REVOKE` commands in general are aimed at much higher-end systems where there are multiple users who have varying access levels to data in the database. In the SQLite model, the application is the main user and has access to the entire database. Access in this model is defined at the application level—specifically, what applications have access to the database file.

In addition to what is listed here, there is a page on the SQLite wiki devoted to reporting unsupported SQL. It is located at `www.sqlite.org/cvstrac/wiki?p=UnsupportedSql`.

Who Should Read This Book

SQLite has many uses and therefore draws a wide and diverse audience. Whether you are a programmer, web developer, systems administrator, or just casual user looking to learn about relational databases, this book aims to help you understand and get the most out of your particular use for SQLite.

SQLite is a terrific database to start on if you are new to relational databases. This book will help you not only get started with SQLite but also become a competent user of SQL. It will also provide you with a good foundation with which to move on to larger relational systems and explore more advanced features and topics.

For programmers, this book assumes only that you know the programming language in which you intend to use SQLite and the basics of relational theory. Furthermore, it does more than document APIs. If anything, that is the least of what it does, because API documentation only illustrates how an interface works. As with any database, you have to have some idea of how that database works internally to get the most out of it. Every database has unique architectural aspects, specific relational features, and important limitations, all of which good programmers learn about and take into consideration when writing their code. SQLite, though simple and straightforward, is no exception. As a programmer, you need to know something about how it processes data internally to get it to work well with your application. This book shows you how. It covers the API and explores how it works in relation to SQLite's architecture, allowing C programmers, web developers, and scriptwriters alike to write more informed code. This helps you better understand not only what SQLite can do but also what it can't. Your knowledge of the architecture will tell you better than any list of rules when SQLite is or isn't a good fit for what you are trying to accomplish. You'll know if, when, and where you need to consider another approach.

One of the most important aims in this book is to teach concepts over recipes—to adequately address both how and why. By the time you're done, you will have both a conceptual and practical understanding of how something works. To that end, this book includes both theoretical material intermixed with many figures and examples designed specifically to illustrate the topics at hand and provide real-world value.

How This Book Is Organized

This book is divided into three parts: SQLite the database, SQLite the programming library, and reference material. The database aspects of SQLite are covered in Chapters 2, 3, and 4. The programming aspects of SQLite are covered in Chapters 5–8. A brief chapter outline is as follows:

Chapter 1, "Introducing SQLite," introduces the main features of SQLite, its origin and history, and the scope and objectives of this book (you are reading this now!)

Chapter 2, "Getting Started," covers how to obtain and use SQLite. It illustrates how to get SQLite in binary and source form, as well as how to compile and build it on a variety of platforms. It explains how to use the SQLite command-line utility to create and work with databases.

Chapter 3, "SQL for SQLite," provides some background behind SQL. It introduces many of the basic SQL commands, such as those for creating tables and selecting and working with data. This chapter acts as both an overview for SQLite's own approach to SQL and a quick introduction for those needing a refresher on SQL itself.

Chapter 4, "Advanced SQL in SQLite," continues from Chapter 3 with coverage of more complex queries and explores every aspect of the remaining commands in SQLite's SQL implementation.

Chapter 5, "SQLite Design and Concepts," lays the groundwork for programming with SQLite. It illustrates the SQLite API, its architecture, and how the two work in relation to one another. It addresses topics such as transactions and locking. It provides programmers of all languages with a clear understanding of how SQLite works internally and what to keep in mind when writing programs that use it.

Chapter 6, "The Core C API," covers the part of the SQLite C API related to executing queries. With sections on connecting to databases, executing queries, obtaining data, managing transactions, and tracing your code, this chapter covers all parts of the API related to query and data processing.

Chapter 7, "The Extension C API," covers the remaining part of the C API devoted to customizing and extending SQLite. SQLite provides facilities for implementing user-defined SQL functions, aggregates, and collations. This chapter illustrates example implementations of these features and provides practical examples of their use.

Chapter 8, "Language Bindings for SQLite," provides a concise introduction to SQLite programming in a range of popular languages such as Perl, Python, Ruby, Java, and PHP.

Chapter 9, "SQLite for Apple iPhone and iPad," focuses on how to use SQLite when developing for Apple's mobile platforms.

Chapter 10, "SQLite for Google Android devices," covers how SQLite is baked in to the Android platform and how to develop with SQLite for the explosion of devices that now run Android.

Chapter 11, "SQLite Internals," explores the inner workings of SQLite. It is a high-level overview of the source code and provides a glimpse into how the major subsystems are implemented. This provides programmers with a deeper understanding of SQLite's design decisions, assumptions, and trade-offs, as well as a point of departure for developers who want to work on SQLite.

DATABASE EXAMPLES

The example databases accompanying this book are available online and can be downloaded from the Apress website (`www.apress.com`). Each database is in SQL format, and you can simply follow the procedures covered in Chapter 2 to create them using the SQLite command-line program. The example databases are further explained and illustrated as they are introduced in this book.

The source code for all examples is also available online. All examples compile and run on Windows, Linux, and Unix. For each example, makefiles are included for Linux and Unix environments, and Visual C++ projects have been created for Windows users.

Additional Information

The SQLite website has a wealth of information, including the official documentation, mailing lists, wiki, and other general information. It is located at `www.sqlite.org`. The SQLite community is very helpful, and you may find everything you need on SQLite's mailing list. Additionally, SQLite's author offers professional training and support for SQLite, which includes custom programming (porting to embedded platforms, etc.), and enhanced versions of SQLite, which include native encryption and extremely small versions optimized for embedded applications. You can find more information at `www.hwaci.com/sw/sqlite/prosupport.html`.

Summary

SQLite is not to be confused with other larger databases like Oracle or PostgreSQL. Whereas dedicated relational databases such as these are electronic juggernauts, SQLite is a digital Swiss Army knife. Whereas large-scale dedicated relational databases are designed for thousands of users, SQLite is designed for thousands of uses. It is more than a database. Although a tool in its own right, it is a tool for making tools as well. It is a true utility, engineered to enable you—the developer, user, or administrator—to quickly and easily shape those disparate piles of data into order, and manipulate them to your liking with minimal effort.

SQLite is public domain software. Free. You can do anything with it or its source code you like. No licenses, no install programs, no restrictions. Just copy and run. It is also portable, well tested, and reliable. It has a clean, modular design that helps keep the system simple, easy to develop, and easy to debug. In addition to good design, it has good testing. There is more code written to test SQLite than there is SQLite code to test. It should not be too surprising, then, that SQLite has proven itself to be a solid, reliable database over its five-year history.

Finally, SQLite is fun. At least we think so, and we hope that you will find it equally useful and enjoyable.

If you have any comments or suggestions on this book or its examples, please feel free to contact the authors at `sqlitebook@gmail.com` (Michael) or `grantondata@gmail.com` (Grant).

CHAPTER 2

Getting Started

It's very easy to begin using SQLite no matter what operating system you are using. For the vast majority of users, you can be up and running with SQLite in less than five minutes, regardless of experience. This chapter covers everything you need to know in order to install SQLite and work with databases. By the time we are done, you will have a working knowledge of where to obtain SQLite software or source code and how to install or compile it on multiple platforms. You'll be working with new SQLite databases and creating tables, views, and indexes that you can query, back up, and restore.

You will learn everything you need to know about managing SQLite databases, including how to create, view, and examine their contents. Finally, you will be introduced to several tools with which to work with SQLite in various environments. This chapter does include some examples that use SQL to introduce the SQLite command-line program. If you are not yet familiar with SQL, the examples will be easy enough that you should still be able to follow them without much trouble. We take an in-depth look at SQL in detail in Chapters 3 and 4.

Where to Get SQLite

The SQLite website (`www.sqlite.org`) provides both precompiled binaries of SQLite as well as source code. Binaries are available for popular platforms including Mac OS X, Windows, and Linux.

There are several binary packages to choose from, each of which is specific to a particular way of using SQLite. The binary packages are as follows:

sqlite3 command-line program (CLP): This version of the SQLite command-line program has the database engine compiled in with statically linked dependencies, acting as a self-contained, stand-alone program. This provides a convenient way to work with SQLite databases from the command line without having to worry about whether the SQLite shared library is installed on your system or located in the right place. If you run on of the contemporary Linux distributions or OS X, you almost certainly already have this installed.

SQLite shared library (DLL or "so"): This is the SQLite database engine packaged into a shared object (so) or Windows dynamic link library (DDL). Use this with programs that dynamically link to SQLite. This form makes it easier to upgrade SQLite without having to recompile the software that depends on it. Note that there is currently no precompiled `.dylib` shared library for Mac OS X.

SQLite Analyzer: The Analyzer tool is useful for inspecting many qualities and statistics about a given SQLite database file, including things such as physical data distribution and layout within the file, relative fragmentation, internal data size and free space, and other measures. This information is very useful for performance optimization work.

Tcl extension: This binary incorporates the Tcl language extensions with the SQLite core. This allows you to connect to SQLite from within the Tcl language. Tcl and C are the two language bindings provided by the SQLite team themselves. Other language bindings are covered in Chapter 8.

SQLite's source code is provided in a variety of forms for convenience and to target different platforms. The main differences in the source distributions concern whether the code is amalgamated into a single source file and header or left in constituent files. There is also a source package incorporating the Tcl Extension Architecture (TEA) for those who want to compile SQLite with Tcl bindings.

The SQLite amalgamated source appears twice on the website—once as a zip file and again as a gzipped tarball. The source code itself does not differ between these source distributions. The zip file is generally recommended and in particular is the only source readily suitable for compilation on Windows (unless you prefer going to laborious effort on that platform). The zip file source distribution benefits from already having certain preprocessing and code generation performed on the code by build tools like GNU `autoconf`. This means Windows users don't need to hunt down Windows-specific equivalents for these tools. The gzipped tarball is targeted to users of Linux, Unix flavors, Mac OS X, BSD variants, and so on. These platforms include in their normal distributions the expected tool chain to do much of the preprocessing work, as well as normal compilation, so users of those platforms can adopt this source package in the knowledge they need not hunt down additional tools to compile SQLite.

SQLite on Windows

Whether you are using SQLite as an end user, writing programs that use SQLite, or using it as a learning platform for relational theory and SQL, SQLite can be installed on Windows with a minimum of fuss. In this section, we will cover all the options, from installing the available binary packages to building everything from source using the most popular compilers. We start with the easy things first and progress to things more technically challenging.

Getting the Command-Line Program

The SQLite command-line program (hereafter referred to as the CLP) is by far the easiest way to get started using SQLite. Follow these steps to obtain the CLP:

1. Open your favorite browser, and navigate to the SQLite home page at `www.sqlite.org`.
2. Click the download link at the top of the page. This will take you to the download page.

3. Scroll down to the section Precompiled Binaries For Windows, where you will find filenames along the lines of `sqlite-3_x_y_z.zip`, where `x`, `y`, and `z` are the minor-version numbers. There should be a comment beside it that reads "A command-line program for accessing and modifying SQLite databases." Download this file to your Windows environment.

4. Unzip the file, either using Windows built-in support for zip files or using your favorite third-party compression tool such as WinZip or 7-Zip. The zip file should contain a single file named `sqlite3.exe`. You can start using the file from the location in which you unzipped it, or if you would like to run the CLP from any directory in the Windows shell, you need to copy it to a folder that is in your Windows system path. A suitable default that should work on all versions of Windows is the `\windows\system32` folder on your root partition (`C:\ for most systems`).

■ **Note** If you don't know your Windows system path value, you can find it in several ways. From a command prompt, you can execute this command:

```
echo %PATH%
```

This will return a string of directories separated by semicolons—this is your path value. For the graphically inclined, you can obtain the same information from the Control Panel. Click Start ➤ Control Panel. Choose the System icon. In the resulting dialog box, select the Advanced tab, and click the Environmental Variables button. In the System Variables list in the lower half of the dialog box, double-click the Path entry. This will open the Edit System Variables dialog box, in which the Values text box contains your path list. You can add additional directories to this path if you like by simply appending a semicolon to the end of the line and typing the new path.

---

5. Open a command shell. You can do this in different ways. Using the Windows Start menu, select Start ➤ Run. Type `cmd` in the Open drop-down box, and click OK (Figure 2-1). This will open a Windows command shell. If this doesn't work, try selecting Start ➤ All Programs ➤ Accessories ➤ Command Prompt.

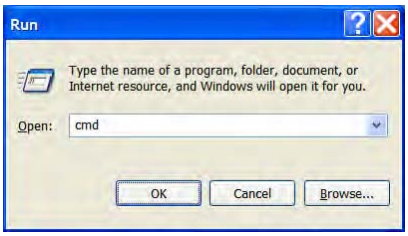


Figure 2-1. *Opening a Windows command shell*

6. Within the shell, type `sqlite3` on the command line, and press Enter. This should bring up a SQLite command prompt. (If you get an error, then the `sqlite3.exe` executable has not been copied to a folder in your system path. Recheck your path, and place a copy of the program somewhere within it.) When the SQLite shell appears, type `.help` on the command line. This will display a list of commands with their associated descriptions similar to the one in Figure 2-2. Type `.exit` to exit the program. You now have a working copy of the SQLite CLP installed on your system.

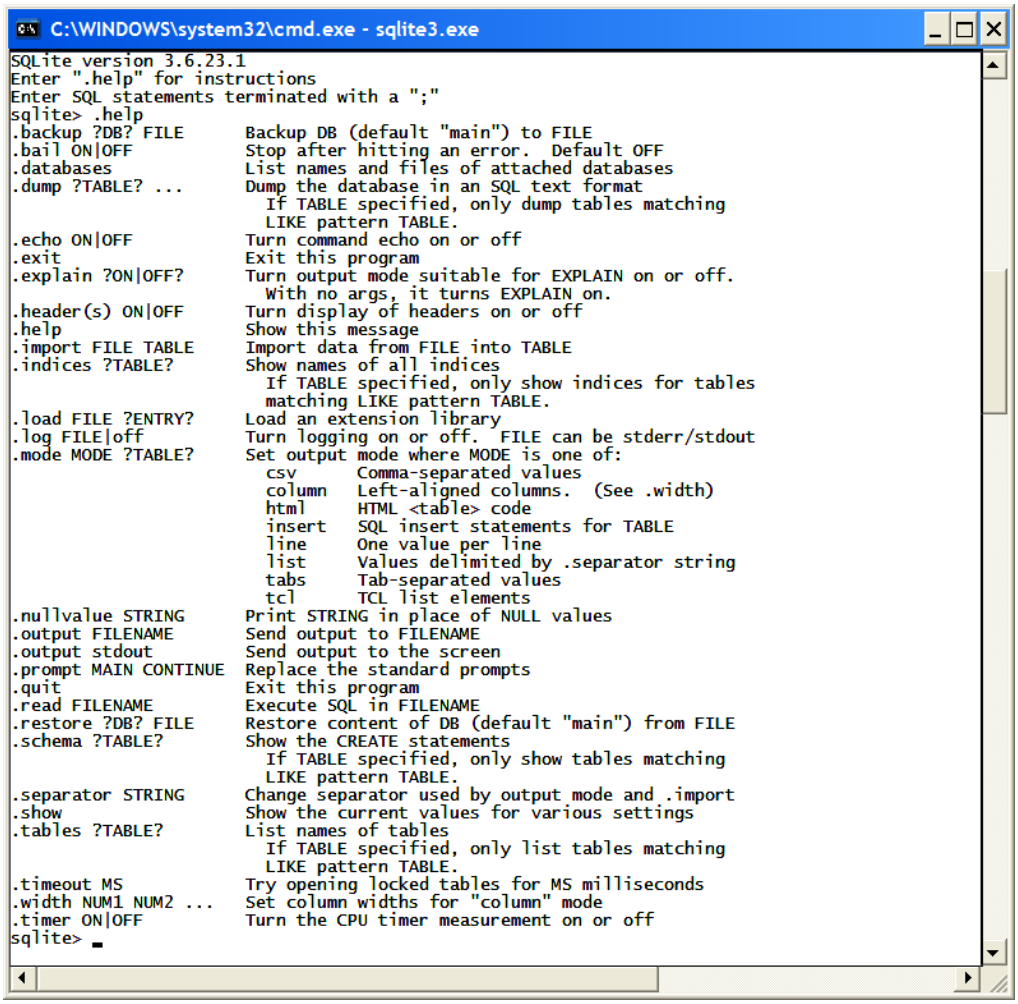


Figure 2-2. *The SQLite shell on Windows*

Download from Wow! eBook <www.wowebook.com>

If you are especially eager to work with SQLite at this point, you may want to skip ahead to the section "The CLP in Shell Mode." The next few sections are geared to developers who want to write programs that use SQLite.

Getting the SQLite DLL

The SQLite DLL is used for software compiled to link dynamically to SQLite. This means that the application will load the DLL at runtime when SQLite features are required, instead of embedding the SQLite code in the application itself. Software that uses SQLite in this fashion typically includes its own copy of the SQLite DLL and installs it automatically with the software.

If you plan to develop with SQLite, using the DLL is probably the easiest way to start. You can obtain the SQLite DLL as follows:

1. Go to the SQLite home page, `www.sqlite.com`, and choose the download link at the top of the page. This will take you to the download page.
2. On the download page, find the section Precompiled Binaries For Windows.
3. Locate the DLL zip file. This file will have the description "This is a DLL of the SQLite library without the TCL bindings." The filename will have the form `sqlitedll-3_x_y_z.zip`, where `x`, `y`, and `z` are the minor versions. If you want Tcl support included, select the file with the name of the form `tclsqlitedll-3_x_y_z.zip`.
4. Download and unzip the file. The extracted contents should include two files: the actual DLL file (`sqlite3.dll`) together with another file called `sqlite3.def`. The SQLite DLL provided here is thread safe, because it was compiled with the `THREADSAFE` preprocessor flag defined. This allows you to use SQLite features in multithreaded applications; you can therefore use this DLL in multithreaded programs, performing simultaneous actions in multiple threads safely.

To use the DLL, it needs to either be in the same folder with programs that use it, be placed somewhere in the system's path (see the note on the Windows System path in the previous section), or follow Window's dynamic library loading rules for locations searched for DLLs.

If you want to write programs that use the SQLite DLL, you will need to create an import library with which to link your programs. This is quite simple to do using the `sqlite3.def` file mentioned earlier. If you are using C++ in Microsoft Visual Studio, open the command prompt, change the directory to the SQLite distribution, and simply run the following command:

```
LIB /DEF:sqlite3.def
```

You should see normal library generation output similar to the following example:

```
C:\sqlite>lib /DEF:sqlite3.def
Microsoft (R) Library Manager Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

LINK : warning LNK4068: /MACHINE not specified; defaulting to X86
   Creating library sqlite3.lib and object sqlite3.exp
```

This will generate an import library named `sqlite3.lib` and an exports file named `.exp`. If you are using MinGW (see the section "Building SQLite with MinGW" later in this chapter), run this command:

```
dlltool --def sqlite3.def --dllname sqlite3.dll --output-lib sqlite3.lib
```

The `dlltool` for MinGW will also create an import library called `sqlite3.lib` with which you can link your programs. By linking your programs to this import library, they will load and use the SQLite DLL upon execution.

Compiling the SQLite Source Code on Windows

Building SQLite from source within Windows is straightforward. Depending on the compiler you are using and what you are trying to achieve, there are several approaches to compiling SQLite. The most common scenarios on Windows include using Microsoft Visual C++, MinGW, or Eclipse (which often uses MinGW for C++ work). We'll cover each of these here. You can find information about how to compile SQLite with other compilers on the SQLite wiki (`www.sqlite.org/cvstrac/wiki?p=HowToCompile`).

The Stable Source Distribution

You can obtain stable versions of SQLite's source code in zip files from the SQLite website. Bleeding-edge versions can be obtained from the Fossil distributed source control system maintained by the SQLite team. Unless you are familiar with Fossil, using the source distribution is the easiest way to go. To download a stable source distribution, follow these steps:

1. Go to the SQLite website, `www.sqlite.org`. Follow the download link, which will take you to the download page.
2. On the download page, find the Source Code section.
3. The first file listed is the recommended zip file containing the source code and suitable for compilation for Windows. The file will be `named sqlite-amalgamation-3-x_y_z.zip`, where `x`, `y`, and `z` (and possibly further minor digits) are the minor-version numbers.

■ **Note** The zip archive and the other tarballs on the download page differ slightly in their contents. Although they contain identical source code, the SQLite distribution uses some POSIX build tools (`sed`, `awk`, and so on) to dynamically generate some C source code in the build process. These build tools are not available by default on Windows systems. Therefore, the zip source archive includes all the preprocessing and generated code as a matter of convenience to Windows users who lack the build support infrastructure of Linux, Unix, Mac OS X, and so on. This is why Windows users should use the zip archives rather than the tarballs on the download page. It is still possible to build the tarballs on Windows, but you need the requisite build tools from a source such as Cygwin, MinGW, GnuWin32, or similar.

---

4. Extract/unzip the file to a directory of your choosing. The extracted contents will be the complete SQLite version 3 source code, suitable for Windows.

Anonymous Fossil Source Control

If you want to play with the latest features or participate in SQLite development, then retrieving SQLite from the project's Fossil source control system makes the most sense. Fossil provides the same kind of anonymous access you'll likely have seen in other source control systems, like CVS, Subversion, Perforce, Git, and so on. Fossil allows you to maintain the absolutely latest version of the SQLite source code by accessing the source straight from your browser! If you want, you can keep your copy of the code synced up to the day, hour, or minute to stay current with changes as they are committed. Thus, if you see an important bug fix or feature posted that you want to take advantage of, you can have Fossil sync and recompile your copy of the code. Fossil even includes an autosync feature to guarantee you stay at the forefront of any source changes.

Obtaining SQLite from Fossil on Windows couldn't be easier. Like SQLite binaries, Fossil is distributed as a single statically linked file. You can download Fossil from the project's home page, `www.fossil-scm.org`. The download page includes up-to-the-minute releases for Windows, as well as Mac OS X and Linux. Unzip the Fossil archive to a location on your path (see our previous discussion on determining your `PATH` environment variable). Next, make a new directory to house your Fossil source extracts, or choose an existing directory if you prefer.

You can do most of your source control work with Fossil in a browser—Fossil acts as its own HTTP server to handle all the server-side tasks. The fastest way to get your initial check-out of the SQLite source code is to use your newly downloaded `fossil.exe` binary to clone one of the SQLite online repositories. Open a command prompt, and change directories to the new or existing directory you've chosen for your SQLite Fossil source control repository. Then at the command prompt, issue the Fossil `clone` command, as shown here:

```
C:\sqlite\fossil.exe clone http://www.sqlite.org/src sqlite.fossil
                Bytes      Cards  Artifacts     Deltas
Send:              49          1          0          0
Received:     1499917      32606          0          0
Send:           10025        225          0          0
Received:      132364        239          7        193
...
Send:            4655         99          0          0
Received:     1698453        125         29         69
Total network traffic: 1379539 bytes sent, 17512520 bytes received
Rebuilding repository meta-data...
32605 (100%)...
project-id: 2ab58778c2967968b94284e989e43dc11791f548
server-id:  38f0c0987054889a5d2c0b4f27b370e9e8632a16
admin-user: fuzz (password is "******")
```

The format of the command is `fossil <action> <Fossil Repository URL> <your repository name>`. In this case, the action was `clone`, meaning to take a complete copy of the SQLite repository. We provide one of the SQLite Fossil repositories described on the SQLite website, in this case `www.sqlite.org/src`. Lastly, we provide a name of our choosing, which will be the name of the file Fossil creates to contain and manage our copy of the source. This can be any name you like that satisfies Windows file-naming conventions. We choose `sqlite.fossil` to be a useful name reminding us what the file contains and what uses it.

■ **Note** We are targeting the current "head" of the source control tree for SQLite with the example given. This means you would be compiling from the latest and greatest source. That's a two-edged sword of course, giving you the absolute latest developments but not the benefit of a "stable" release. Feel free to choose a stable branch if you prefer.

You are now ready to launch and use the Fossil web interface. From your command line, just run the next command (substitute your own directory and local repository filename as appropriate).

```
C:\sqlite\fossil.exe ui sqlite.fossil
```

You should see output from Fossil indicating it has successfully launched its HTTP server and is listening for connections. This should very quickly disappear behind your favorite browser, which Fossil will have automatically opened at the home of your local SQLite repository, as shown in Figure 2-3.

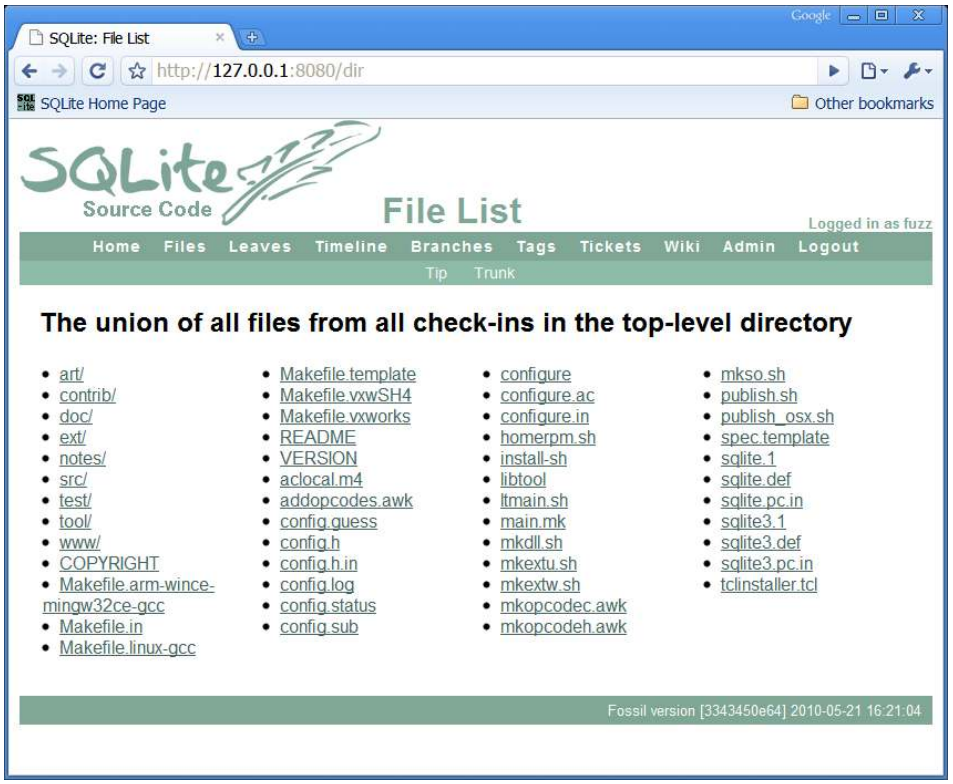


Figure 2-3. *Your local SQLite Fossil source control system, complete with up-to-date source*

You are now up and running with Fossil. From here, you'll obviously want to start with checking out the code so that you can start work. That's just as easy as most other Fossil actions. First, from your shell, create a directory for the source, and then change into that directory.

```
mkdir c:\sqlite\src
cd c:\sqlite\src
```

Now you can tell Fossil to check out the current incarnation of the source, using the `open` command:

```
c:\sqlite\fossil.exe open c:\sqlite\sqlite.fossil
```

You'll see all the source files for the SQLite project fly by on the screen. When completed, around 15MB of files are downloaded, and you have the source ready with which to work.

Building the SQLite DLL with Microsoft Visual C++

To build the SQLite DLL from source using Visual C++, follow these steps:

1. Start Visual Studio. Create a new DLL project within the unpacked SQLite source directory. Do this by selecting File ➤ New ➤ Project. Under Project Types (Figure 2-4), select Visual C++ Projects, and then select Win32. Choose the Win32 Project template. In the Location text box, enter the folder name that contains your SQLite source folder. In this example, it would be `C:\sqlite`. In the Name text box, enter the name of the folder containing the SQLite source code—`src` in this example. This will create the Visual C++ project inside the existing SQLite source folder (`C:\sqlite\src`). Click OK.

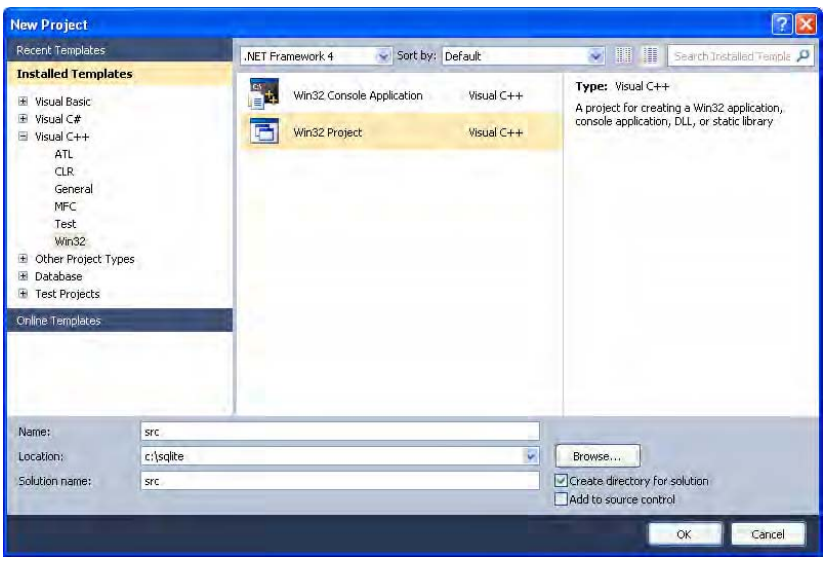


Figure 2-4. *Creating a new Visual C++ project*

2. Next, the Win 32 Application Wizard will automatically open (Figure 2-5). Choose Application Settings, and set the application type to DLL. Be sure to select the Empty Project box. Click Finish, and this will create a blank DLL project.

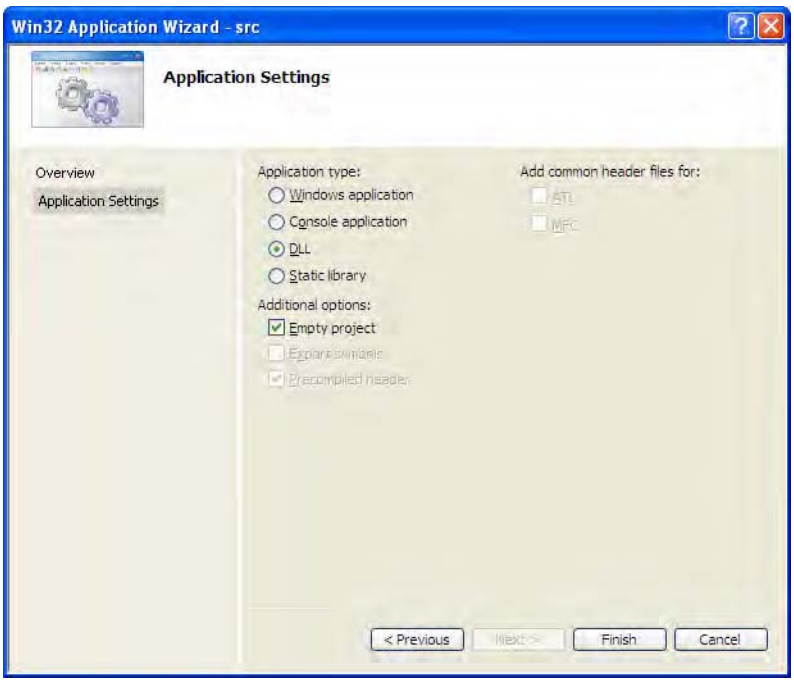


Figure 2-5. *The Win32 Application Wizard*

3. Add SQLite source files and headers to the project. Select Project ➤ Add Existing Item. Add all `.c` and `.h` files in the directory except for two files: `tclsqlite.c` and `shell.c`. (The first is for Tcl support; the second is for creating the SQLite CLP, neither of which we want in this case.)

4. Specify an export or a module definition (`.def`) file. This file defines what symbols (or functions) to export (make visible) to programs that link to the library. SQLite's source distribution is kind enough to include such a file (`sqlite3.def`) for this very purpose. Also within the Property Pages dialog box, select All Configurations in the Configuration drop-down box (Figure 2-6). Then click the Linker folder, and click the Input submenu. In the Module Definition File property page, type `sqlite3.def`. You are now ready to build the DLL.

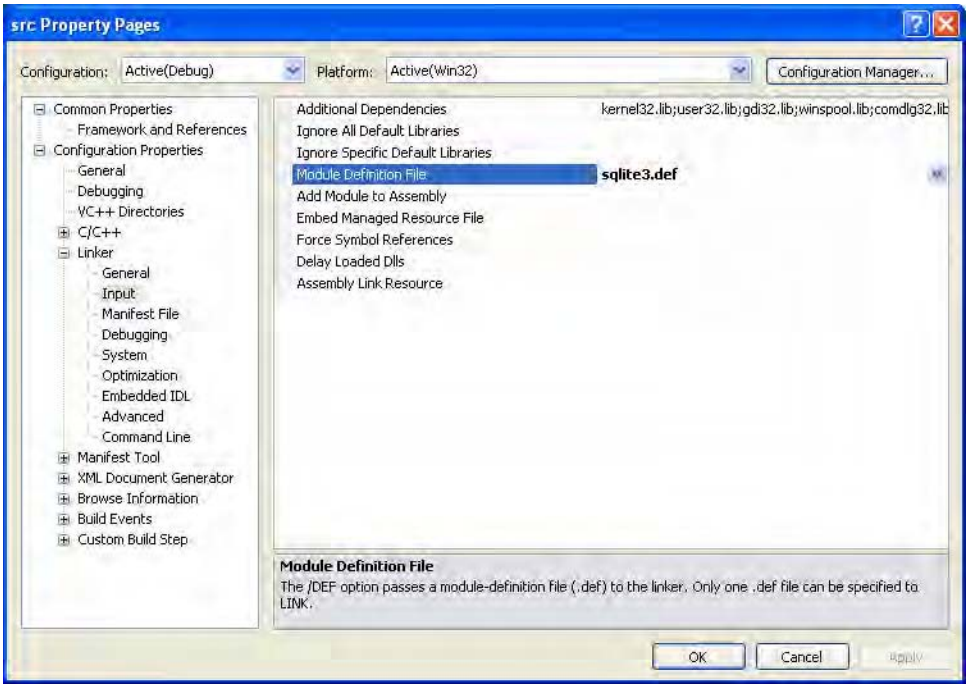


Figure 2-6. *Project properties*

5. From the main menu, select Build ➤ Build sqlite to build the DLL.
6. Once you have built the DLL, be sure to create the import library, as described in the section "Getting the SQLite DLL."

Building a Dynamically Linked SQLite Client with Visual C++

The binary for a static CLP is available on the SQLite website, but what if you want a version that uses the SQLite DLL? To build such a version in Visual C++, do the following:

■ **Note** Many of the steps are very similar to the process of building a DLL, mentioned earlier—you may want to use some of the figures listed there for reference.

---

1. From the main menu, select File ➤ New ➤ Project. Under Project Types, select Visual C++ Projects, and then select Win32. Choose the Win32 Project template. Name the project (`shell`, for example), and click OK.

2. After this, the Win 32 Application Wizard will automatically open. Choose Application Settings, set the application type to Console Application, and be sure to check the Empty Project box. Click Finish to create a blank executable project.
3. Next you want to add the SQLite CLP source file. Select Project ➤ Add Existing Item. In the dialog box that appears, add the source file `shell.c`.
4. Tell Visual C++ to link against the SQLite DLL. Select Project ➤ Properties. In the dialog box that appears, select All Configurations in the Configuration drop-down box. Next select the Linker folder. Select the Input submenu, and within the Additional Dependencies property page, add `sqlite3.lib`. You are now ready to build the program. Note that the SQLite DLL needs to be either in the same directory as the command-line program or in the Windows system path.

■ **Note** If you build the SQLite DLL with threading enabled or you obtain the DLL from the SQLite website, you need to use the multithreaded Microsoft C runtime library DLL when building the CLP. To do this, refer to the second half of step 4 in "Building the SQLite DLL with Microsoft Visual C++." It contains two informative figures that make it easy to set this option.

Building SQLite with MinGW

MinGW (`www.mingw.org`) is a fork of the Cygwin project, and it provides a nice distribution of the GNU Compiler Collection (GCC) for Windows. It also includes freely available Windows-specific header files and libraries that you can use to create native Windows programs that do not rely on any third-party C runtime DLLs. Put simply, it is a free C/C++ compiler for Windows, and it's a very good one at that. It is usually used in conjunction with MSYS, which offers a bash-like shell and POSIX environment that makes Unix users feel at home on Windows. Together, the two provide a powerful environment with which to compile and build software on Windows. The two are popular but also can be difficult to stick together by hand. Thankfully, several very good bundles exist to take all the hard work out of deploying them, allowing you to get on with the fun of compiling your own SQLite from source. We recommend the TakeOffGW package, freely available from SourceForge.

To build the SQLite DLL from source with the TakeOffGW distribution of MinGW and MSYS, do the following:

1. Open your favorite browser, and navigate to the SourceForge website: `sourceforge.net`.
2. Search for the TakeOffGW project, or navigate straight to the project page at `sourceforge.net/projects/takeoffgw/`.
3. Download the latest release of the TakeOffGW network installer, usually just called `setup.exe`.

4. After downloading the TakeOffGW network installer, run the `setup.exe` file, and choose the defaults for installation location, package directory, and links to the Internet (remembering any proxy server you may use). When you get to the package selection page, shown in Figure 2-7, be sure to select all the packages available under the MSYS heading. Do this by toggling the MSYS heading itself until the phrase "install" appears.

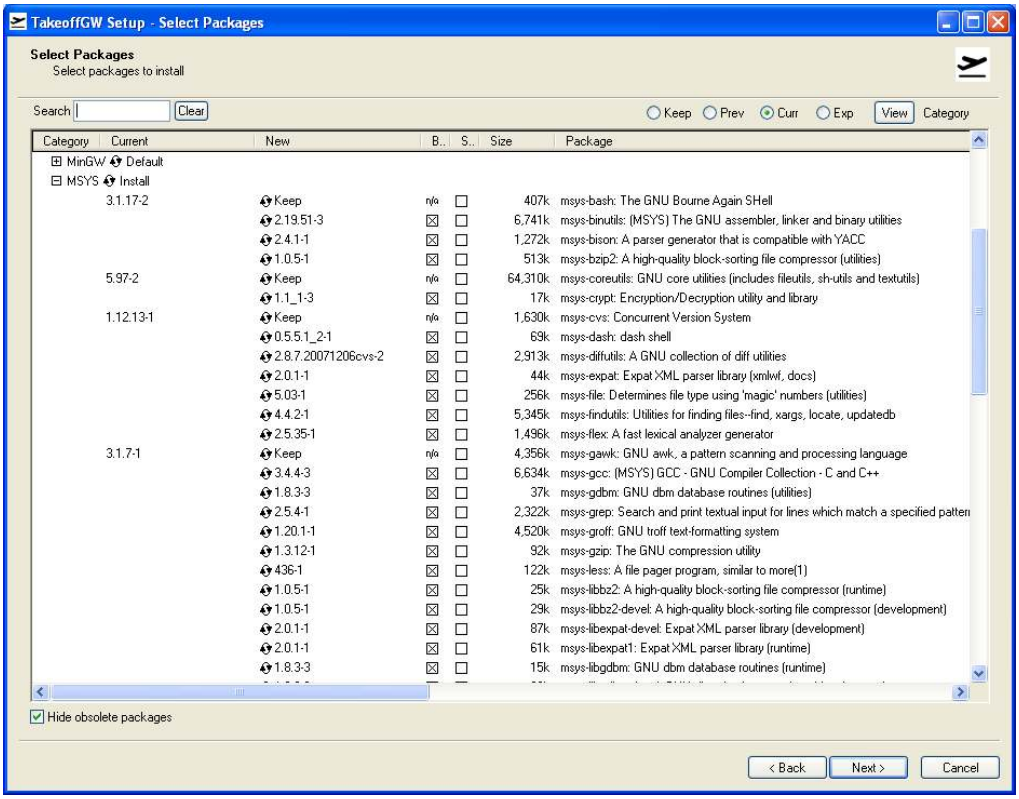


Figure 2-7. *TakeOffGW installation components, with MSYS selected for install*

5. Click Next to continue, and TakeOffGW will download and install all the components required.

6. Download the Linux/Unix SQLite source code distribution. Navigate to `www.sqlite.org`, and click the download link. On the download page, find the Source Code section. The file you are looking for is the source distribution in tarballs form, which should have a name of the form `sqlite-amalgamation-3.x.y.z.tar.gz`, where `x`, `y`, and `z` (and possibly further numbers) are the minor-version numbers (at the time of this writing, the current filename is `sqlite-amalgamation-3.6.23.1.tar.gz`). Download the tarball, and place it in a temporary directory (e.g., `C:\Temp`).
7. TakeOffGW will have placed an icon on your desktop. Double-click that icon to open the environment.
8. Navigate to your temporary directory in which you downloaded the SQLite source distribution. Since this is a Unix-like environment, you will need to use Unix file system conventions. For example, to get to `c:\Temp`, you would type `cd /c/Temp`.
9. Unpack the SQLite tarball. Issue this command: tar -xzvf `sqlite-amalgamation-3.6.23.1.tar.gz`.
10. Move into the unpacked directory:

    ```
    cd sqlite-3.6.32.1
    ```

11. Create the makefile. For the SQLite binary, run this:

    ```
    ./configure
    ```

12. Build the source:

    ```
    make
    ```

You will see `configure` and `make` scroll many lines of status output to the screen. When `make` completes, you now have a functional SQLite binary. To use it easily, add the TakeOffGW `bin` directory to your path. If you followed the defaults prompted by the TakeOffGW network installer, this will be the `c:\cygwin\bin` directory. You can now use Windows Explorer or the command prompt, navigate to the temporary directory, and run `sqlite3.exe`. You now have a working SQLite CLP.

Download from Wow! eBook <www.wowebook.com>

SQLite on Linux, Mac OS X, and Other POSIX Systems

SQLite compiles and builds identically on systems such as Linux, Mac OS X, FreeBSD, NetBSD, OpenBSD, Solaris, and others—known historically as POSIX systems (though this is now a less useful term, because systems as diverse as mainframes and Microsoft Windows have been POSIX compliant for more than a decade). SQLite binaries can be obtained in a variety of ways depending on the particular operating system.

Binaries and Packages

If you are using Mac OS 10.4 (Tiger) or greater, you already have SQLite installed on your system. If not, there are several routes you can take to install it. The easiest way is to use one of the following Mac-specific package management systems, all of which include packages or ports for SQLite:

MacPorts: MacPorts is probably the most popular source of open source software, tools, and encompassing package management features with Mac users today. It includes the absolute latest SQLite distributions, including additional packages such as the Tcl bindings, and so on.

Fink: Fink is a Debian-based package management system that uses Debian utilities such as `dpkg`, `dselect`, and `apt-get`, in addition to its own utility—`fink`. You can download Fink from `http://fink.sourceforge.net`. With Fink, it is possible to install straight from precompiled binaries. No compilation step is needed.

BSD users will have no trouble installing SQLite either. FreeBSD, OpenBSD, and NetBSD all have packages and/or ports for SQLite, all of which are very easy to install. Each distribution has ports for very recent versions of SQLite 3.6.*x*.

Solaris 10 uses SQLite as part of the OS, originally shipping with version 2, but if your system is patched regularly, contemporary Solaris 10 platforms are patched to version 3.6.20.

As mentioned earlier, binaries for Linux are available directly from the SQLite website. The download page on SQLite's website provides the following binaries:

Statically linked command-line program: The filename is of the form `sqlite3-3.`*x.y.z*`.bin.gz`, where `x`, `y`, and `z` (and possibly more digits) are the minor-version numbers.

Shared library: Two forms of the shared library exist. One form includes the Tcl bindings; the other does not. The description next to each highlights which shared library source is which. Note that the shared libraries provided are not thread safe. If you need a thread-safe version, you will have to compile the library from source. See the section "Compiling SQLite from Source" for more details.

SQLite Analyzer: This is a command-line program that provides detailed information about the contents of a SQLite database. You'll find information on this program in the section "Getting Database File Information."

Probably the best way to get new versions of SQLite and/or its source for your Linux platform is from the relevant package repositories for your distribution. Red Hat, CentOS, Fedora, and other Red Hat derivatives can use `yum` to search and find various SQLite packages. Debian-based distributions (including Ubuntu) will have no trouble getting up-to-date versions of SQLite. SQLite 3 packages are available online in both Ubuntu and Debian repositories, among others. Use `apt` or Synaptic to search and retrieve the packages of your choice.

Compiling SQLite from Source

Compiling SQLite from source on Linux, Mac OS X, the BSD flavors, or other Unix systems follows very closely the MinGW/TakeOffGW instructions given earlier for the Windows platform (actually, it is more the other way around; MinGW installation apes Linux source installation!). To build SQLite on these systems, you need to ensure that you have the GNU Compiler Collection (GCC) installed, including Autoconf, Automake, and Libtool. Most of the systems already discussed include all of these by default. With this software in place, you can build SQLite by doing the following:

1. Download the Linux/Unix SQLite tarball (source code) from the SQLite website. At the time of this writing, the current version is `sqlite-amalgamation-3.6.23.1.tar.gz`. Place it in a directory (e.g., `/tmp`).
2. Navigate to your build directory:

   ```
   cd /tmp
   ```
3. Unpack the SQLite tarball:

   ```
   tar -xzvf sqlite-amalgamation-3.6.23.1.tar.gz
   ```
4. Move into the unpacked directory:

   ```
   cd sqlite-3.6.23.1
   ```
5. Create the makefile:

   ```
   ./configure
   ```
6. Other options, such as the installation directory, are also available. For a complete list of configure options, run this:

   ```
   ./configure --help
   ```
7. Build the source:

   ```
   make
   ```
8. As root, install:

   ```
   make install
   ```

You now have a functional SQLite installation on your system. If you have GNU Readline installed on you system, the CLP should be compiled with Readline support. Test it by running it from the command line:

```
root@linux # sqlite3
```

This will invoke the CLP using an in-memory database. Type `.help` for a list of shell commands. Type `.exit` to close the application, or press Ctrl+D.

The Command-Line Program

The SQLite CLP is the most common means you can use to work with and manage SQLite databases. It operates the same way on all platforms, so learning how to use it ensures you will always have a common and familiar way to manage your databases. The CLP is really two programs in one. It can be run in shell mode acting as an interactive query processor, or it can run from the command line to perform various administration tasks.

The CLP in Shell Mode

Open a shell, and change directories to some temporary folder—say `C:\Temp` if you are on Windows or `/tmp` if you're in Unix. If you're happy working from a more permanent location, you can create a `sqlite` directory—e.g., `c:\sqlite` under Windows or `/sqlite` for Linux or Unix. This will be your current working directory. All files you create in the course of working with the shell will be created in this directory.

■ **Note** If you need a refresher on how to get to the Windows command prompt, refer to step 5 in the "Getting the Command-Line Program" section earlier in this chapter.

To invoke the CLP as in shell mode, type `sqlite3` from a command line, followed by an optional database name. If you do not specify a database name, SQLite will use an in-memory database (the contents of which will be lost when the CLP exits).

Using the CLP as an interactive shell, you can issue queries, obtain schema information, import and export data, and perform many other database tasks. The CLP will consider any statement issued as a query, except for commands that begin with a period (`.`). These commands are reserved for specific CLP operations, a complete list of which can be obtained by typing `.help`, as shown here:

```
fuzzy@linux:/tmp$ sqlite3
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .help
```

```
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail ON|OFF           Stop after hitting an error.  Default OFF
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL text format
                         If TABLE specified, only dump tables matching
                         LIKE pattern TABLE.
.echo ON|OFF           Turn command echo on or off
.exit                  Exit this program
.explain ?ON|OFF?      Turn output mode suitable for EXPLAIN on or off.
                         With no args, it turns EXPLAIN on.
.genfkey ?OPTIONS?     Options are:
                         --no-drop: Do not drop old fkey triggers.
                         --ignore-errors: Ignore tables with fkey errors
                         --exec: Execute generated SQL immediately
                       See file tool/genfkey.README in the source
                       distribution for further information.
.header(s) ON|OFF      Turn display of headers on or off
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
```

```
.indices ?TABLE?       Show names of all indices
                         If TABLE specified, only show indices for tables
                         matching LIKE pattern TABLE.
.load FILE ?ENTRY?     Load an extension library
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                         csv      Comma-separated values
                         column   Left-aligned columns.  (See .width)
                         html     HTML <table> code
                         insert   SQL insert statements for TABLE
                         line     One value per line
                         list     Values delimited by .separator string
                         tabs     Tab-separated values
                         tcl      TCL list elements
.nullvalue STRING      Print STRING in place of NULL values
.output FILENAME       Send output to FILENAME
.output stdout         Send output to the screen
.prompt MAIN CONTINUE  Replace the standard prompts
.quit                  Exit this program
.read FILENAME         Execute SQL in FILENAME
.restore ?DB? FILE     Restore content of DB (default "main") from FILE
.schema ?TABLE?        Show the CREATE statements
                         If TABLE specified, only show tables matching
                         LIKE pattern TABLE.
.separator STRING      Change separator used by output mode and .import
.show                  Show the current values for various settings
.tables ?TABLE?        List names of tables
                         If TABLE specified, only list tables matching
                         LIKE pattern TABLE.
.timeout MS            Try opening locked tables for MS milliseconds
.width NUM1 NUM2 ...   Set column widths for "column" mode
.timer ON|OFF          Turn the CPU timer measurement on or off
.width NUM NUM ...     Set column widths for "column" mode
```

```
sqlite>.exit
```

You can just as easily type `.h` for short. Many of the commands can be similarly abbreviated, such as `.e`—short for `.exit`—to exit the shell.

The CLP in Command-Line Mode

You can use the CLP from the command line for tasks such as importing and exporting data, returning result sets, and performing general batch processing. It is ideal for use in shell scripts for automated database administration. To see what the CLP offers in command-line mode, invoke it from the shell (Windows or Unix) with the `-help` switch, as shown here:

```
fuzzy@linux:/tmp$ sqlite3 -help
```

```
Usage: sqlite3 [OPTIONS] FILENAME [SQL]
FILENAME is the name of an SQLite database. A new database is created
if the file does not previously exist.
OPTIONS include:
   -help                show this message
   -init filename       read/process named file
   -echo                print commands before execution
   -[no]header          turn headers on or off
   -bail                stop after hitting an error
   -interactive         force interactive I/O
   -batch               force batch I/O
   -column              set output mode to 'column'
   -csv                 set output mode to 'csv'
   -html                set output mode to HTML
   -line                set output mode to 'line'
   -list                set output mode to 'list'
   -separator 'x'       set output field separator (|)
   -nullvalue 'text'    set text string for NULL values
   -version             show SQLite version   -init filename       read/process named file
```

The CLP in command-line mode takes the following arguments:

- A list of options (optional)
- A database filename (optional)
- A SQL command to execute (optional)

Most of the options control output formatting except for the `init` switch, which specifies a batch file of SQL commands to process. The database filename is required. The SQL command is optional with a few caveats.

Database Administration

Now you that you've seen how to invoke the CLP both interactively and in command-line mode, it's time to look at some examples of using the CLP for some common, administrative tasks. We'll begin at the beginning, with database creation.

Creating a Database

Let's start by creating a database that we will call `test.db`. From the command line, open the CLP in shell mode by typing the following:

```
sqlite3 test.db
```

Even though we have provided a database name, SQLite does not actually create the database (yet) if it doesn't already exist. SQLite will defer creating the database until you actually create something inside it, such as a table or view. The reason for this is so that you have the opportunity to set various

permanent database settings (such as page size) before the database structure is committed to disk. Some settings such as page size and character encoding (UTF-8, UTF-16, etc.) cannot be changed easily once the database is created, so this is where you have a chance to specify them. We will go with the default settings here, so to actually create the database on disk, we need only to create a table. Issue the following statement from the shell:

```
sqlite> create table test (id integer primary key, value text);
```

Now you have a database file on disk called `test.db`, which contains one table called `test`. This table, as you can see, has two columns:

- A primary key column called `id`, which has the ability to automatically generate values by default. Wherever you define a column of type integer primary key, SQLite will apply an function for the column to create and apply monotonically increasing values. That is, if no value is provided for the column in an `INSERT` statement, SQLite will automatically generate one by finding the next integer value specific to that column.
- A simple text field called `value`.

Let's add a few rows to the table:

```
sqlite> insert into test (id, value) values(1, 'eenie');
sqlite> insert into test (id, value) values(2, 'meenie');
sqlite> insert into test (value) values('miny');
sqlite> insert into test (value) values('mo');
```

Now fetch them back:

```
sqlite> .mode column
sqlite> .headers on
sqlite> select * from test;
```

```
id          value
----------  ----------
1           eenie
2           meenie
3           miny
4           mo
```

The two commands preceding the `select` statement (`.headers` and `.mode`) are used to improve the formatting a little (these commands and others like them are covered later). We can see that our explicit ID values for the first two rows were used. We can also see that SQLite provided sequential integer values for the `id` column for rows 3 and 4, which we did not provide in the `insert` statements. While on the topic of autoincrement columns, you might be interested to know that the value of the last inserted autoincrement value can be obtained using the SQL function `last_insert_rowid()`:

```
sqlite> select last_insert_rowid();
```

```
last_insert_rowid()
-------------------
4
```

Before we quit, let's add an index and a view to the database. These will come in handy in the examples that follow:

```
sqlite> create index test_idx on test (value);
sqlite> create view schema as select * from sqlite_master;
```

To exit the shell, issue the `.exit` command:

```
sqlite> .exit
```

On Windows, you can also terminate the shell by using the key sequence Ctrl+C. On Unix, you can use Ctrl+D.

Getting Database Schema Information

There are several shell commands for obtaining information about the contents of a database. You can retrieve a list of tables (and views) using `.tables [pattern]`, where the optional `[pattern]` can be any pattern that the SQL `like` operator understands (we cover `like` in Chapter 3 if you are unfamiliar with it). All tables and views matching the given pattern will be returned. If no pattern is supplied, all tables and views are returned:

```
sqlite> .tables
```

```
schema test
```

Here we see our table named `test` and our view named `schema`. Similarly, indexes for a given table can be printed using `.indices [table name]`:

```
sqlite> .indices test
```

```
test_idx
```

Here we see the index we created earlier on `test`, called `test_idx`. The SQL definition or data definition language (DDL) for a table or view can be obtained using `.schema [table name]`. If no table name is provided, the SQL definitions of all database objects (tables, indexes, views, and triggers) are returned:

```
sqlite> .schema test
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE INDEX test_idx on test (value);
```

```
sqlite> .schema
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE VIEW schema as select * from sqlite_master;
CREATE INDEX test_idx on test (value);
```

More detailed schema information can be had from SQLite's principal system view, sqlite_master. This view is a simple system catalog of sorts. Its schema is described in Table 2-1.

Table 2-1. *SQLite Master Table Schema*

Name	Description
`type`	The object's type (table, index, view, trigger)
`name`	The object's name
`tbl_name`	The table the object is associated with
`rootpage`	The object's root page index in the database (where it begins)
`sql`	The object's SQL definition (DDL)

Querying `sqlite_master` for our current database returns the following (don't forget to use the `.mode column` and `.headers on` commands first to manually set the column format and headers):

```
sqlite> .mode column
sqlite> .headers on
sqlite> select type, name, tbl_name, sql from sqlite_master order by type;
```

```
type        name        tbl_name    sql
----------  ----------  ----------  -----------------------------------
index       test_idx    test        CREATE INDEX test_idx on test (value)
table       test        test        CREATE TABLE test (id integer primary
view        schema      schema      CREATE VIEW schema as select * from s
```

We see a complete inventory of `test.db` objects: one table, one index, and one view, each with their respective original DDL creation statements.

There are few additional commands for obtaining schema information through SQLite's `PRAGMA` commands, `table_info`, `index_info`, and `index_list`, which are covered in Chapter 4.

■ **Tip** Don't forget that most command-line tools like the SQLite CLP keep a history of the commands that you execute. To rerun a previous command, you can hit the Up Arrow key to scroll through your previous commands. On Windows, you can also hit `F7` in any command prompt window to see a scrollable list of the commands you have entered.

Exporting Data

You can export database objects to SQL format using the `.dump` command. Without any arguments, `.dump` will export the entire database as a series of DDL and data manipulation language (DML) commands, suitable for re-creating the database objects and the data contained therein. If you provide arguments, the shell interprets them as table names or views. Any tables or views matching the given arguments will be exported. Those that don't are simply ignored. In shell mode, the output from the `.dump` command is directed to the screen by default. If you want to redirect output to a file, use the `.output [filename]` command. This command redirects all output to the file `filename`. To restore output back to the screen, simply issue `.output stdout`. So, to export the current database to a file `file.sql`, you would do the following:

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
```

This will create the file `file.sql` in your current working directory if it does not already exist. If a file by that name does exist, it will be overwritten.

By combining redirection with SQL and the various shell formatting options (covered later), you have a great deal of control over exporting data. You can export specific subsets of tables and views in various formats using the delimiter of your choice, which can later be imported using the `.import` command described next.

Importing Data

There are two ways to import data, depending on the format of the data in the file to import. If the file is composed of SQL, you can use the `.read` command to execute the commands contained in the file. If the file contains comma-separated values (CSV) or other delimited data, you can use the `.import [file][table]` command. This command will parse the specified file and attempt to insert it into the specified table. It does this by parsing each line in the file using the pipe character (|) as the delimiter and inserting the parsed columns into the table. Naturally, the number of parsed fields in the file should match up with the number of columns in the table. You can specify a different delimiter using the `.separator` command. To see the current value set for the separator, use the `.show` command. This will show all user-defined settings for the shell, among them the current default separator:

```
sqlite> .show
```

```
     echo: off
  explain: off
  headers: on
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
```

The `.read` command is the way to import files created by the `.dump` command. Using `file.sql` created earlier as a backup, we can drop the existing database objects (the `test` table and `schema` view) and re-import it as follows:

```
sqlite> drop table test;
sqlite> drop view schema;
sqlite> .read file.sql
```

Formatting

The shell offers a number of formatting options to help you and make your results and output neat and tidy. The simplest are `.echo`, which echoes the last run command after issuing a command, and .headers, which includes column names for queries when set to `on`. The text representation of `NULL` can be set with `.nullvalue`. For instance, if you want `NULLs` to appear as the text string NULL, simply issue the command `.nullvalue NULL`. By default, this presentation value is an empty string.

The shell prompt can be changed using `.prompt [value]`:

```
sqlite> .prompt 'sqlite3> '
sqlite3>
```

Result data can be formatted several ways using the `.mode` command. The current options are `csv`, `column`, `html`, `insert`, `line`, `list`, `tabs`, and `tcl`, each of which is helpful in different ways. The default is `.list`. For instance, list mode displays results with the columns separated by the default separator. Thus, if you wanted to dump a table in a CSV format, you could do the following:

```
sqlite3> .output file.csv
```

Download from Wow! eBook <www.wowebook.com>

```
sqlite3> .separator ,
sqlite3> select * from test;
sqlite3> .output stdout
```

The contents of `file.csv` now appear as shown next.

```
1,eenie
2,meenie
3,miny
4,mo
```

Actually, since there is a CSV mode already defined in the shell, it is just as easy to use it in this particular example instead:

```
sqlite3> .output file.csv
sqlite3> .mode csv
sqlite3> select * from test;
sqlite3> .output stdout
```

The results obtained are the same. The difference is that CSV mode will wrap field values with double quotes, whereas list mode (the default) does not.

Exporting Delimited Data

Combining the previous three sections on exporting, importing, and formatting data, we now have an easy way to export and import data in delimited form. For example, to export only the rows of the `test` table whose `value` fields start with the letter *m* to a file called `test.csv` in comma-separated values, do the following:

```
sqlite> .output text.csv
sqlite> .separator ,
sqlite> select * from test where value like 'm%';
sqlite> .output stdout
```

If you want to then import this CSV data into a similar table with the same structure as the `test` table (call it `test2`), do the following:

```
sqlite> create table test2(id integer primary key, value text);
sqlite> .import text.csv test2
```

The CLP, therefore, makes it easy to both import and export text-delimited data to and from the database.

Performing Unattended Maintenance

So far, you've seen the CLP used interactively to perform tasks such as creating a database and exporting data. However, you don't always want to be tied to your seat, executing CLP commands one at a time. Instead, you can use the command mode to run CLP commands in batches. You can then use your operating system's built-in scheduler to schedule those batches to run whenever you need them.

■ **Note** You are free to invoke the CLP from the command line interactively. Any time you have a sequence of commands that you want to invoke routinely, it's useful to use the command-line approach.

There are actually two ways to invoke the CLP in command-line mode. The first is to provide a SQL command, or a SQLite shell command as well, such as `.dump` and `.schema`. Any valid SQL or SQLite shell command will do. SQLite will execute the specified command, print the result to standard output, and exit. For example, to dump the `test.db` database from the command line, issue the following command:

```
sqlite3 test.db .dump
```

To make it useful, we should redirect the output to a file:

```
sqlite3 test.db .dump > test.sql
```

The file `test.sql` now contains the complete human-readable set of DDL and DML statements for the database `test.db`. Similarly, to select all records for the `test` table, issue this:

```
sqlite3 test.db "select * from test"
```

The second way to invoke the CLP in command-line mode is to redirect a file as an input stream. For instance, to create a new database `test2.db` from our database dump `test.sql`, do the following:

```
sqlite3 test2.db < test.sql
```

The CLP will read the file as standard input and then process and apply all SQL commands within it to the `test2.db` database file.

Another way to create a database from the `test.sql` file is to use the `init` option and provide the `test.sql` as an argument:

```
sqlite3 -init test.sql test3.db
```

The CLP will process `test.sql`, create the `test3.db` database, and then go into shell mode. Why? The invocation included no SQL command or input stream. To get around this, you need to provide a SQL command or SQLite shell command. For example:

```
sqlite3 -init test.sql test3.db .exit
```

The `.exit` command prompts the CLP to run in command-line mode and does as little as possible. All things considered, redirection is perhaps the easiest method for processing files from the command line.

Backing Up a Database

Backing up a database can be done in two ways, depending on the type of backup you desire. A SQL dump is perhaps the most portable form for keeping backups. The standard way to generate one is using

the CLP `.dump` command, as shown in the previous section. From the command line, this is done as follows:

```
sqlite3 test.db .dump > test.sql
```

Within the shell, you can redirect output to an external file, issue the command, and restore output to the screen as follows:

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
sqlite> .exit
```

Likewise, importing a database is most easily done by providing the SQL dump as an input stream to the CLP:

```
sqlite3 test.db < test.sql
```

This assumes that `test.db` does not already exist. If it does, then things may still work if the contents of `test.sql` are different from those of `test.db`. You will of course get errors if `test.sql` contains objects that already reside within `test.db` or contains data that violates primary key or foreign key constraints (though see the `PRAGMA` discussion in subsequent chapters on how to finesse this behavior).

Making a binary backup of a database is little more than a file copy. One small operation you may want to perform beforehand is a database vacuum, which will free up any unused space created from deleted objects. This will provide you with a smaller resulting file from the binary copy:

```
sqlite3 test.db vacuum
cp test.db test.backup
```

As a general rule, binary backups are not as portable as SQL backups. On the whole, SQLite does have good backward compatibility and is binary compatible across all platforms for a given database format. However, for long-term backups, it is always a good idea to use SQL form. If size is an issue, SQL format (raw text) usually yields a good compression ratio.

■ **Caution** No matter how good you think your chosen backup approach is, remember you are only as good as your last successful *restore*. Test your restore procedure if you need to rely on it—otherwise, you'll be remembered for one failed restore, regardless of how many successful backups you took.

Finally, if you've worked with other databases, "dropping" a database in SQLite, like binary backups, is a simple file operation: you simply delete the database file you want to drop.

Getting Database File Information

The primary means by which to obtain logical database information, such as table names, DDL statements, and so on, is using the `sqlite_master` view, which provides detailed information about all objects contained in a given database.

If you want information on the physical database structure, you can use a tool called SQLite Analyzer, which can be downloaded in binary form from the SQLite website. SQLite Analyzer provides detailed technical information about the on-disk structure of a SQLite database. This information includes a detailed breakdown of database, table, and index statistics for individual objects and in aggregate. It provides everything from database properties such as page size, number of tables, indexes, file size, and average page density (utilization) to detailed descriptions of individual database objects. Following the report is a detailed list of definitions explaining all terms used within the report. A partial output of `sqlite_analyzer` is as follows:

```
fuzzy@linux:/tmp$ sqlite3_analyzer test.db
```

```
/** Disk-Space Utilization Report For test.db
*** As of 2010-May-07 20:26:23

Page size in bytes.................... 1024
Pages in the whole file (measured).... 3
Pages in the whole file (calculated).. 3
Pages that store data................. 3          100.0%
Pages on the freelist (per header).... 0            0.0%
Pages on the freelist (calculated).... 0            0.0%
Pages of auto-vacuum overhead......... 0            0.0%
Number of tables in the database...... 2
Number of indices..................... 1
Number of named indices............... 1
Automatically generated indices....... 0
Size of the file in bytes............. 3072
Bytes of user payload stored.......... 26           0.85%

*** Page counts for all tables with their indices ********************

TEST.................................. 2           66.7%
SQLITE_MASTER......................... 1           33.3%

*** All tables and indices ***********************************************

Percentage of total database.......... 100.0%
Number of entries..................... 11
Bytes of storage consumed............. 3072
Bytes of payload...................... 235          7.6%
Average payload per entry............. 21.36
Average unused bytes per entry........ 243.00
Maximum payload per entry............. 72
Entries that use overflow............. 0            0.0%
Primary pages used.................... 3
Overflow pages used................... 0
```

```
Total pages used..................... 3
Unused bytes on primary pages........ 2673        87.0%
Unused bytes on overflow pages....... 0
Unused bytes on all pages............ 2673        87.0%

*** Table TEST and all its indices ***********************************

Percentage of total database.........  66.7%
Number of entries..................... 8
Bytes of storage consumed............. 2048
Bytes of payload...................... 60           2.9%
Average payload per entry............. 7.50
Average unused bytes per entry........ 243.00
Maximum payload per entry............. 10
Entries that use overflow............. 0            0.0%
Primary pages used.................... 2
Overflow pages used................... 0
Total pages used..................... 2
Unused bytes on primary pages........ 1944        94.9%
Unused bytes on overflow pages....... 0
Unused bytes on all pages............ 1944        94.9%
```

SQLite Analyzer is provided in binary form on the SQLite website for Linux, Mac OS X, and Windows. SQLite Analyzer can be built from the source using the Unix makefile provided. From the build directory, issue the following command:

```
make sqlite3_analyzer
```

You must, however, have Tcl support configured in the build settings because SQLite Analyzer uses the Tcl extension to perform most of its work. Refer to "Compiling SQLite from Source" for more information.

Other SQLite Tools

There are many other open source and commercial programs available with which to work with SQLite. Good graphical, cross-platform tools include the following:

SQLite Database Browser (`http://sqlitebrowser.sourceforge.net`): Allows users to manage databases, tables, and indexes, as well as import and export them. Users can interactively run SQL queries and inspect the results, as well as examine a log of all SQL commands issued. It recently received a major upgrade to version 2.

SQLiteman (`http://www.sqliteman.com`): A cross-platform program that's targeted at people managing and administering SQLite databases. It allows for general management of databases, tables, indexes, and triggers, as well as other common management tasks.

SQLiteManager (`www.sqlabs.net/sqlitemanager.php`): A commercial software package designed for working with and administering SQLite. Users can manage database objects, execute queries, and save SQL, as well as create reports with flexible report templates. It includes its own high-level scripting language to complement its SQL capabilities.

These are just the cross-platform tools. Many more tools are available that can be used with just about any programming, end-user, or management environment. You can find more information on such packages on the SQLite wiki (`www.sqlite.org/cvstrac/wiki?p=ManagementTools`).

Summary

No matter what platform you work on, SQLite is easy to install and build. Windows, Mac OS X, and Linux users can obtain binaries directly from the SQLite website. Users of many other operating systems can also obtain binaries using their native—or even third-party—package systems.

The common way to work with SQLite across all platforms is using the SQLite command-line program (CLP). This program operates as both a command-line tool and an interactive shell. You can issue queries and do essential database administration tasks such as creating tables, indexes, and views as well as exporting and importing data. SQLite databases are contained in single operating system files, so doing things like backups are very simple—just copy the file. For long-term backups, however, it is always best to dump the database in SQL format, because this is portable across SQLite versions.

In the next few chapters, you will be using the CLP to explore SQL and the database aspects of SQLite. We will start with the basics of using SQL with SQLite in Chapter 3 and move to more advanced SQL in Chapter 4.

CHAPTER 3

SQL for SQLite

This chapter is an introduction to using SQL with SQLite. SQL makes up a huge component of any discussion about databases, and SQLite is no different. Whether you're a novice or pro with SQL, this chapter offers comprehensive coverage. Even if you've never used SQL before, you should have no trouble with the material covered in this chapter. If you've also avoided the relational model that underpins SQLite and other RDBMSs, don't fret. We'll augment our discussion of SQL with just enough of the relational concepts to make things clear, without getting bogged down in the theory.

SQL is the sole (and almost universal) means by which to communicate with a relational database. It is the workhorse devoted to information processing. It is designed for structuring, reading, writing, sorting, filtering, protecting, calculating, generating, grouping, aggregating, and in general managing information.

SQL is an intuitive, user-friendly language. It can be fun to use and is quite powerful. One of the fun things about SQL is that regardless of whether you are an expert or a novice, it seems that you can always continue to learn new ways of doing things (for better or worse).

The goal of this chapter is to teach you to use SQL *well*—to expose you to good techniques and perhaps a few cheap tricks along the way. As you can already tell from the table of contents, SQL for SQLite is such a large topic that we've split its discussion into two halves. We cover the core `select` statement first and then move on to other SQL statements in Chapter 4. But by the time you are done, you should be well equipped to put a dent in any database.

■ **Note** Even with two chapters, we really only have space to focus on SQL's use in SQLite. For a much wider and in-depth coverage of SQL in general, we recommend a book like *Beginning SQL Queries* by Claire Churcher.

The Example Database

Before diving into syntax, let's get situated with the obligatory example database. The database used in this chapter (and the rest of this book, for that matter) consists of all the foods in every episode of *Seinfeld*. If you've ever watched *Seinfeld,* you can't help but notice a slight preoccupation with food. There are more than 400 different foods mentioned in the 180 episodes of its history (according to fan data spread far and wide on the Internet). That's more than two new foods every show. Subtract commercial time, and that's virtually a new food introduced every 10 minutes.

As it turns out, this preoccupation with food works out nicely because it makes for a database that illustrates all the requisite concepts of SQL and of using SQL in SQLite. Figure 3-1 shows the database tables for our sample database.

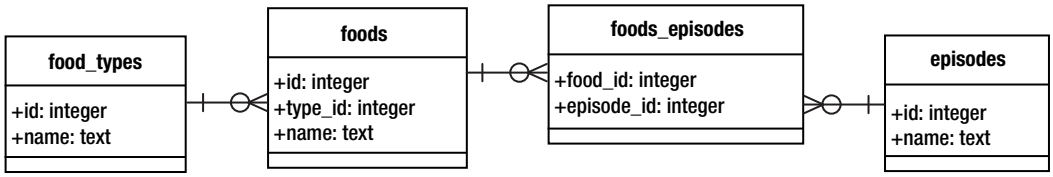


Figure 3-1. *The Seinfeld food database*

The database schema, as defined in SQLite, is defined as follows:

```
create table episodes (
  id integer primary key,
  season int,
  name text );

create table foods(
  id integer primary key,
  type_id integer,
  name text );

create table food_types(
  id integer primary key,
  name text );

create table foods_episodes(
  food_id integer,
  episode_id integer );
```

The main table is `foods`. Each row in `foods` corresponds to a distinct food item, the name of which is stored in the `name` attribute. The `type_id` attribute references the `food_types` table, which stores the various food classifications (e.g., baked goods, drinks, or junk food). Finally, the `foods_episodes` table links foods in `foods` with the episodes in `episodes`.

Installation

The food database is located in the examples zip file accompanying this book. It is available on the Apress website (`www.apress.com`) in the Source Code section. To create the database, first locate the `foods.sql` file in the root directory of the unpacked zip file. To create a new database from scratch from the command line, navigate to the examples directory, and run the following command:

```
sqlite3 foods.db < foods.sql
```

This will create a database file called `foods.db`. If you are not familiar with how to use the SQLite command-line program, refer to Chapter 2.

Running the Examples

For your convenience, all the SQL examples in the chapter are available in the file `sql.sql` in the root directory of the examples zip file. So instead of typing them by hand, simply open the file and locate the SQL you want to try.

A convenient way to run the longer queries in this chapter is to copy them into your favorite editor and save them in a separate file, which you can run from the command line. For example, copy a long query in `test.sql` to try. You simply use the same method to run it as you did to create your database earlier:

```
sqlite3 foods.db < test.sql
```

The results will be printed to the screen. This also makes it easier to experiment with these queries without having to retype them or edit them from inside the SQLite shell. You just make your changes in the editor, save the file, and rerun the command line.

For maximum readability of the output, you may want to put the following commands at the beginning of the file:

```
.echo on
.mode column
.headers on
.nullvalue NULL
```

This causes the command-line program to 1) echo the SQL as it is executed, 2) print results in column mode, 3) include the column headers, and 4) print nulls as `NULL`. The output of all examples in this chapter is formatted with these specific settings. Another option you may want to set for various examples is the `.width` option, which sets the respective column widths of the output. These vary from example to example.

For better readability, the examples are presented in two different formats. For short queries, we'll show the SQL and output as it would be shown from within the SQLite shell. For example:

```
sqlite> select *
   ...> from foods
   ...> where name='JujyFruit'
   ...> and type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

Occasionally, as in the previous example, we take the liberty of adding an extra line between the command and its output to improve readability here on the printed page. For longer queries, we show just the SQL in code format separated from the results by gray lines, as in the following example:

```
select f.name name, types.name type
from foods f
inner join (
  select *
  from food_types
  where id=6) types
on f.type_id=types.id;
```

```
name                        type
--------------------------  -----
Generic (as a meal)         Dip
Good Dip                    Dip
Guacamole Dip               Dip
Hummus                      Dip
```

Syntax

SQL's declarative syntax reads a lot like a natural language. Statements are expressed in the imperative mood, beginning with the verb describing the action. Following it are the subject and predicate, as illustrated in Figure 3-2.

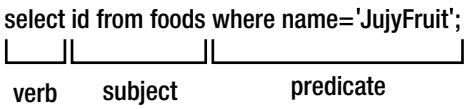


Figure 3-2. *General SQL syntax structure*

As you can see, it reads like a normal sentence. SQL was designed specifically with nontechnical people in mind and was thus meant to be very simple and easy to understand.

Part of SQL's ease of use comes from its being (for the most part) a *declarative* language, as opposed to an imperative language such as C or Perl. A declarative language is one in which you describe *what* you want, whereas an imperative language is one in which you specify *how* to get it. For example, consider the process of ordering a cheeseburger. As a customer, you use a declarative language to articulate your order. That is, you simply declare to the person behind the counter what you want:

Give me a double meat Whataburger with jalapeños and cheese, hold the mayo.

Download from Wow! eBook <www.wowebook.com>

The order is passed back to the chef who fulfills the order using a program written in an imperative language—the recipe. He follows a series of well-defined steps that must be executed in a specific order to create the cheeseburger per your (declarative) specifications:

1. Get ground beef from the third refrigerator on the left.
2. Make a patty.
3. Cook for three minutes.
4. Flip.
5. Cook three more minutes.
6. Repeat steps 1–5 for second patty.
7. Add mustard to top bun.
8. Add patties to bottom bun.
9. Add cheese, lettuce, tomatoes, onions, and jalapeños to burger.
10. Combine top and bottom buns, and wrap in yellow paper.

As you can see, declarative languages tend to be more succinct than imperative ones. In this example, it took the declarative burger language (DBL) one step to materialize the cheeseburger, while it took the imperative chef language (ICL) 10 steps. Declarative languages do more with less. In fact, SQL's ease of use is not far from this example. A suitable SQL equivalent to our imaginary DBL statement shown earlier might be something along the lines of this:

```
select burger
from kitchen
where patties=2
and toppings='jalopenos'
and condiment != 'mayo'
limit 1;
```

Pretty simple. As we've mentioned, SQL was designed to be a user-friendly language. In the early days, SQL was targeted specifically for end users for tasks such as ad hoc queries and report generation (unlike today where it is almost exclusively the domain of developers and database administrators).

Commands

SQL is made up of commands. Commands are typically terminated by a semicolon, which marks the end of the command. For example, the following are three distinct commands:

```
select id, name from foods;
insert into foods values (null, 'Whataburger');
delete from foods where id=413;
```

■ **Note** The semicolon is used by SQLite as a command terminator, signaling the end of the command to be processed. Command terminators are most commonly associated with interactive programs designed for letting users execute queries against a database. The semicolon is a common SQL command terminator across many systems, though some use `\g` or even the word `go`.

Commands, in turn, are composed of a series of *tokens*. Tokens can be literals, keywords, identifiers, expressions, or special characters. Tokens are separated by white space, such as spaces, tabs, and newlines.

Literals

Literals, also called *constants*, denote explicit values. There are three kinds: string literals, numeric literals, and binary literals. String literals are one or more alphanumeric characters surrounded by single quotes. Here are some examples:

```
'Jerry'
'Newman'
'JujyFruit'
```

Although SQLite supports delimiting string values with single or double quotes, we strongly suggest you only use single quotes. This is the SQL standard and will save you many hours of grief should you encounter a system that strictly enforces this. If single quotes are part of the string value, they must be represented as two successive single quotes. For example, Kenny's chicken would be expressed as follows:

```
'Kenny''s chicken'
```

Numeric literals are represented in integer, decimal, or scientific notation. Here are some examples:

```
-1
3.142
6.0221415E23
```

Binary values are represented using the notation `x'0000'`, where each digit is a hexadecimal value. Binary values must be expressed in multiples of 2 hexadecimal values (8 bits). Here are some examples:

```
x'01'
X'0fff'
x'0F0EFF'
X'0f0effab'
```

Keywords and Identifiers

Keywords are words that have a specific meaning in SQL. These include words such as `select`, `update`, `insert`, `create`, `drop`, and `begin`. *Identifiers* refer to specific objects within the database, such as tables or indexes. Keywords are reserved words and may not be used as identifiers. SQL is case insensitive with respect to keywords and identifiers. The following are equivalent statements:

```
SELECT * from foo;
SeLeCt * FrOm FOO;
```

By default, string literal values are case sensitive in SQLite, so the value `'Mike'` is not the same as the value `'mike'`.

Comments

Comments in SQLite are denoted by two consecutive hyphens (`--`), which comment the remaining line, or by the multiline C-style notation (`/* */`), which can span multiple lines. Here's an example:

```
-- This is a comment on one line
/* This is a comment spanning
   two lines */
```

Again, unless you have a strong reason to use C-style notation, we'd recommend sticking with the SQL standard of two consecutive hyphens in your SQL scripts for SQLite.

Creating a Database

Tables are the natural starting point to kick off your exploration of SQL in SQLite. The table is the standard unit of information in a relational database. Everything revolves around tables. Tables are composed of rows and columns. And although that sounds simple, tables bring along with them all kinds of other concepts, which can't be nicely summarized in a few tidy paragraphs. In fact, it takes almost the whole chapter. So, what we are going to do here is the two-minute overview of tables—just enough for you to create a simple table and get rid of it if you want to. And once we have that out of the way, all the other parts of this chapter will have something to build on.

Creating Tables

Like the relational model, SQL consists of several parts. It has a structural part, for example, which is designed to create and destroy database objects. This part of the language is generally referred to as a *data definition language* (DDL). Similarly, it has a functional part for performing operations on those objects (such as retrieving and manipulating data). This part of the language is referred to as a *data manipulation language* (DML). Creating tables falls under DDL, the structural part.

You create a table with the `create table` command, which is defined as follows:

```
create [temp] table table_name (column_definitions [, constraints]);
```

The `temp` or `temporary` keyword creates a *temporary table*. This kind of table is, well, temporary—it will last only as long your session. As soon as you disconnect, it will be destroyed (if you haven't already destroyed it manually). The brackets around the `temp` denote that it is an optional part of the command. Whenever you see any syntax in brackets, it means that the contents within them are optional. Furthermore, the pipe symbol (|) denotes an alternative (think of the word or). Take, for example, the following syntax:

```
create [temp|temporary] table … ;
```

This means that either the `temp` or `temporary` keyword may be optionally used. You could say `create temp table foo…`, or you could say `create temporary table foo…`. In this case, they mean the same thing.

If you don't create a temporary table, then `create table` creates a base table. The term *base table* refers to a table that is a named, persistent table in the database. This is the most common kind of table. In general, the term *base table* is used to differentiate tables created by `create table` from *system tables* and other table-like objects such as views.

The minimum required information for `create table` is a table name and a column name. The name of the table, given by `table_name`, must be unique among all other identifiers. In the body, `column_definitions` consists of a comma-separated list of column definitions composed of a name, a *domain*, and a comma-separated list of *column constraints*. A type, sometimes referred to as a *domain*, is synonymous with a data type in a programming language. It denotes the type of information that is stored in the column.

There are five native types in SQLite: `integer`, `real`, `text`, `blob`, and `null`. All of these types are covered in the section called "Storage Classes" later in this chapter. *Constraints* are constructs that control what kind of values can be placed in the table or in individual columns. For instance, you can ensure that only unique values are placed in a column by using a `unique` constraint. Constraints are covered in the section "Data Integrity."

The `create table` command allows you to include a list of additional column constraints at the end of the command, denoted by `constraints` in the syntax outlined earlier. Consider the following example:

```
create table contacts ( id integer primary key,
                        name text not null collate nocase,
                        phone text not null default 'UNKNOWN',
                        unique (name,phone) );
```

Column `id` is declared to have type `integer` and constraint `primary key`. As it turns out, the combination of this type and constraint has a special meaning in SQLite. `Integer primary key` basically turns the column into an autoincrement column (as explained in the section "Primary Key Constraints" later in this chapter). Column `name` is declared to be of type `text` and has two constraints: `not null` and `collate nocase`. Column `phone` is of type `text` and has two constraints defined as well. After that, there is a table-level constraint of `unique`, which is defined for columns `name` and `phone` together.

This is a lot of information to absorb all at once, but it will all be explained in due course. You can probably appreciate our preamble that warned you how complex tables could become. The only important thing here, however, is that you understand the general format of the `create table` statement.

Altering Tables

You can change parts of a table with the `alter table` command. SQLite's version of `alter table` can either rename a table or add columns. The general form of the command is as follows:

```
alter table table { rename to name | add column column_def }
```

Note that there is some new notation here: {}. Braces enclose a list of options, where one option is required. In this case, we have to use either `alter table table rename…` or `alter table table add column….` That is, you can either rename the table using the `rename` clause or add a column with the `add column` clause. To rename a table, you simply provide the new name given by `name`.

If you add a column, the column definition, denoted by `column_def`, follows the form in the `create table` statement. It is a name, followed by an optional domain and list of constraints. Here's an example:

```
sqlite> alter table contacts
        add column email text not null default '' collate nocase;
sqlite> .schema contacts

create table contacts ( id integer primary key,
                        name text not null collate nocase,
                        phone text not null default 'UNKNOWN',
                        email text not null default '' collate nocase,
                        unique (name,phone) );
```

To view the table definition from within the SQLite command-line program, use the `.schema` shell command followed by the table name. It will print the current table definition. If you don't provide a table name, then `.schema` will print the entire database schema.

Tables can also be created from the results of `select` statements, allowing you to create not only the structure but also the data at the same time. This particular use of the `create table` statement is covered later in the section "Inserting Records."

Querying the Database

The effort spent designing your database and creating your tables all focuses on one purpose: *using* your data. Working with your data is the job of SQL's DML. The core of DML is the `select` command, which has the unique honor of being the sole command for querying the database. `select` is by far the largest and most complex command in SQL. `select` derives many of its operations from relational algebra and encompasses a large portion of it. The capabilities and complexities of the `select` statement are *vast*, even in a streamlined efficient environment like SQLite. Don't let that put you off. SQLite's approach to the `select` command is very logical, and it has a firm grounding in the underlying relational theory that underpins all RDBMS.

Relational Operations

For the theoretically minded, it's useful to think about what select does and why it does it, framed in a conceptual way. In most SQL implementations, including SQLite's, the `select` statement provides for the "relational operations" that source, mix, compare, and filter data. These relational operations are typically divided into three categories:

- Fundamental operations
    - Restriction
    - Projection
    - Cartesian product

    - Union
    - Difference
    - Rename
- Additional operations
    - Intersection
    - Natural join
    - Assign
- Extended operations
    - Generalized projection
    - Left outer join
    - Right outer join
    - Full outer join

The fundamental operations are just that: fundamental. They define the basic relational operations, and all of them (with the exception of rename) have their basis in set theory. The additional operations are for convenience, offering a shorthand way of performing frequently used combinations of the fundamental operations. For instance, intersection can be performed by taking the union of two sets and from that removing via difference the results of two further unions that have each had difference applied for one of the two initial sets. Finally, the extended operations add features to the fundamental and additional operations. For example, the generalized projection operation adds arithmetic expressions, aggregates, and grouping features to the fundamental projection operations. The outer joins extend the join operations and allow additional information and/or incomplete information to be retrieved from the database.

In standard ANSI SQL, `select` can perform every one of these relational operations. These operations map to the original relational operators defined by E. F. Codd in his original work on relational theory (with the exception of `divide`). SQLite supports all the relational operations in ANSI SQL with the exception of right and full outer joins. The types of joins can be constructed using combinations of other fundamental relational operations, so don't fret their absence.

All of these operations are defined in terms of relations or, as they are commonly called, tables! They take one or more relations as their inputs and produce a relation as their output. This allows operations to be strung together into relational expressions. Relational expressions can therefore be created to arbitrary complexity. For example, the output of a select operation (a relation) can be fed as the input to another `select` statement, as follows:

```
select name from (select name, type_id from (select * from foods));
```

Here, the output of the innermost `select` is fed to the next `select`, whose output is in turn fed to the outermost `select`. It is all a single relational expression. Anyone familiar with piping commands in Linux, Unix, or Windows will appreciate this behavior. The output of any `select` statement can be used as input to yet another statement.

select and the Operational Pipeline

The `select` command incorporates many of the relational operations through a series of *clauses* in its syntax. Each clause corresponds to specific relational operation. In SQLite, almost all of the clauses are optional, in fact, to a degree beyond what standard SQL suggests is desirable. As a user of SQL in SQLite, this empowers you to employ only the operations you need to obtain the data you seek.

A very general form of `select` in SQLite (without too much distracting syntax) can be represented as follows:

```
select [distinct] heading
from tables
where predicate
group by columns
having predicate
order by columns
limit count,offset;
```

Each keyword—`from`, `where`, `having`, and so on—is a separate clause. Each clause consists of the keyword followed by arguments (represented in italics). From here on out, we will refer to these clauses simply by name, and you'll be able to recognize our use by the fixed-width font. So, rather than "the `where` clause," we will simply use `where`.

In SQLite, the best way to think of the `select` command is as a pipeline that processes relations. This pipeline has optional processes that you can plug into it as you need them. Regardless of whether you include or exclude a particular operation, the relative operations are always the same. Figure 3-3 shows this order.

The `select` command starts with `from`, which takes one or more input relations; combines them into a single composite relation; and passes it through the subsequent chain of operations.

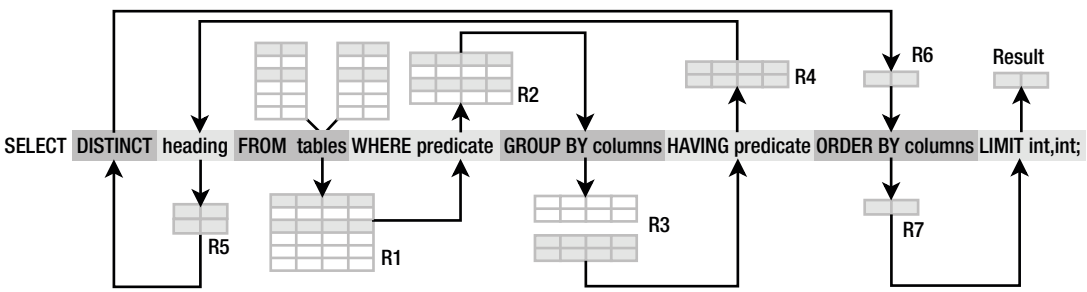


Figure 3-3. *The phases of* `select` *in SQLite*

All clauses are optional with the exception of `select`. You must always provide at least this clause to make a valid `select` command. By far the most common invocation of the `select` command consists of three clauses: `select`, `from`, and `where`. This basic syntax and its associated clauses are as follows:

```
select heading from tables where predicate;
```

The `from` clause is a comma-separated list of one or more tables, views, and so on (represented by the variable `tables` in Figure 3-3). If more than one table is specified, they will be combined to form a single relation (represented by R1 in Figure 3-3). This is done by one of the join operations. The resulting relation produced by `from` serves as the initial material. All subsequent operations will either work directly from it or work from derivatives of it.

The `where` clause filters rows in *R1*. The argument of `where` is a *predicate*, or logical expression, which defines the selection criteria by which rows in *R1* are included in (or excluded from) the result. The selected rows from the `where` clause form a new relation *R2*, as shown in Figure 3-3 above.

In this particular example, *R2* passes through the other operations unscathed until it reaches the `select` clause. The `select` clause filters the columns in *R2*, as depicted in Figure 3-4. Its argument consists of a comma-separated list of columns or expressions that define the result. This is called the *projection list,* or *heading* of the result.

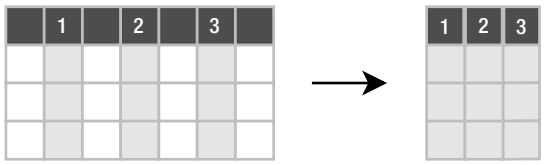


Figure 3-4. *Projection*

Here's a concrete example using our example database:

```
sqlite> select id, name from food_types;
id          name
----------  ----------
1           Bakery
2           Cereal
3           Chicken/Fowl
4           Condiments
5           Dairy
6           Dip
7           Drinks
8           Fruit
9           Junkfood
10          Meat
11          Rice/Pasta
12          Sandwiches
13          Seafood
14          Soup
15          Vegetables
```

There is no `where` clause to filter rows, so all rows in the table `food_types` are returned. The `select` clause specifies all the columns in `food_types`, and the `from` clause does not join tables. The result is an exact copy of `food_types`. As in most SQL implementations, SQLite supports an asterisk (*) as shorthand to mean all columns. Thus, the previous example can just as easily be expressed as follows:

```
select * from food_types;
```

In summary, SQLite's basic `select` processing gathers all data to be considered in the `from` clause, filters rows (restricts) in the `where` clause, and filters columns (projects) in the `select` clause. Figure 3-5 shows this process.

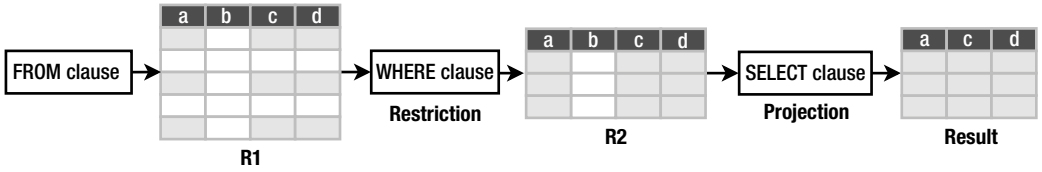


Figure 3-5. *Restriction and projection in* `select`

With this simple example, you can begin to see how a query language in general and SQL in particular ultimately operate in terms of relational operations. There is real math under the hood.

Filtering

If the `select` command is the most complex command in SQL, then the `where` clause is the most complex clause in `select`. But it also does most of the work. Having a solid understanding of its mechanics will most likely bring the best overall returns in your day-to-day use of SQL in SQLite.

SQLite applies the `where` clause to each row of the relation produced by the `from` clause (*R1*). As stated earlier, `where`—a restriction—is a filter. The argument of `where` is a logical predicate. A predicate, in the simplest sense, is just an assertion about something. Consider the following statement:

The dog (subject) is purple and has a toothy grin (predicate).

The dog is the subject, and the predicate consists of the two assertions (color is purple and grin is toothy). This statement may be true or false, depending on the dog.

The subject in the `where` clause is a row. The row is the logical subject. The `WHERE` clause is the logical predicate. Each row in which the proposition evaluates to true is included (or selected) as part of the result (*R2*). Each row in which it is false is excluded. So, the dog proposition translated into a relational equivalent would look something like this:

```
select * from dogs where color='purple' and grin='toothy';
```

The database will take each row in table `dogs` (the subject) and apply the `where` clause (the predicate) to form the logical proposition:

```
This row has color='purple' and grin='toothy'.
```

If it is true, then the row (or dog) is indeed purple and toothy, and it is included in the result. `where` is a powerful filter. It provides you with a great degree of control over the conditions with which to include (or exclude) rows in (or from) the result.

Values

Values represent some kind of data in the real world. Values can be classified by their type, such as a numerical value (1, 2, 3, and so on) or string value ("JujyFruit"). Values can be expressed as *literal values* (an explicit quantity such as 1, 2, 3 or "JujyFruit"), *variables* (often in the form of column names like `foods.name`), expressions (3+2/5), or the results of functions (`count(foods.name)`)—which are covered later.

Operators

An operator takes one or more values as input and produces a value as output. An operator is so named because it performs some kind of operation, producing some kind of result. *Binary operators* are operators that take two input values (or operands). *Ternary operators* take three operands, *unary operators* take just one, and so on.

Operators can be strung together, feeding the output of one operator into the input of another (Figure 3-6), forming value expressions.

Download from Wow! eBook <www.wowebook.com>

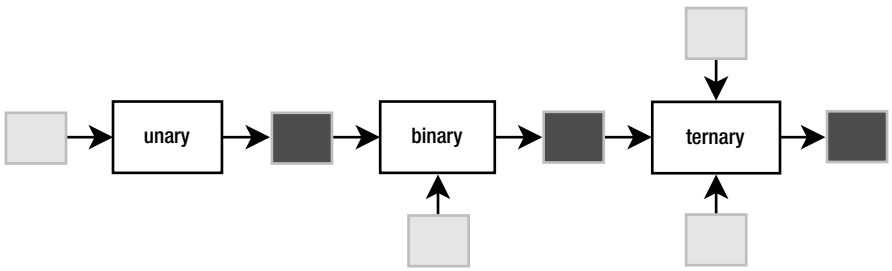


Figure 3-6. Unary, binary, and ternary operators, also displaying pipelining of operations

By stringing operators together, you can create value expressions that are expressed in terms of yet other value expressions to arbitrary complexity. Here's an example:

```
x = count(episodes.name)
y = count(foods.name)
z = y/x * 11
```

Binary Operators

Binary operators are by far the most common of all operators in SQL. Table 3-1 lists the binary operators supported in SQLite by *precedence*, from highest to lowest. Operators in each color group have equal precedence.

Table 3-1. *Binary Operators*

Operator	Type	Action
`||`	String Concatenation	
`*`	Arithmetic Multiply	
`/`	Arithmetic Divi	de
`%`	Arithmetic Modul	us
`+`	Arithmetic	Add
–	Arithmetic Subtra	ct
`<<`	Bitwise Right	shift
`>>`	Bitwise Left	shift
`&`	Logical	And
`|`	Logical Or	
`<`	Relational Les	s than
`<=`	Relational	Less than or equal to
`>`	Relational Grea	ter than
`>=`	Relational	Greater than or equal to
`=`	Relational Equa	l to
`==`	Relational Equa	l to
`<>`	Relational Not	equal to
`!=`	Relational Not	equal to
`IN`	Logical In	
`AND`	Logical	And
`OR`	Logical Or	

Continued

Operator	Type	Action
`IS`	Logical	Equal to
`LIKE`	Relational	String matching
`GLOB`	Relational	Filename matching

Arithmetic operators (for example, addition, subtraction, division) are binary operators that take numeric values and produce a numeric value. *Relational operators* (for example, `>`, `<`, `=`) are binary operators that compare values and value expressions and return a *logical value* (also called a *truth* value), which is either true or false. Relational operators form *logical expressions*, for example:

```
x > 5
1 < 2
```

A logical expression is any expression that returns a truth value. In SQLite, false is represented by the number 0, while true is represented by anything else. For example:

```
sqlite> SELECT 1 > 2;

1 > 2
----------
0

sqlite> SELECT 1 < 2;

1 < 2
----------
1

sqlite> SELECT 1 = 2;

1 = 2
----------
0

sqlite> SELECT -1 AND 1;

-1 AND 1
----------
1
```

Logical Operators

Logical operators (`AND`, `OR`, `NOT`, `IN`) are binary operators that operate on truth values or logical expressions. They produce a specific truth value depending on their inputs. They are used to build more complex logical expressions from simpler expressions, such as the following:

```
(x > 5) AND (x != 3)
(y < 2) OR (y > 4) AND NOT (y = 0)
(color='purple') AND (grin='toothy')
```

The truth value produced by a logical operator follows normal Boolean logic rules, but there is an added twist when considering nulls, which we'll discuss later in the chapter. This is the stuff the `where` clause is made of—using logical operators to answer real questions about the data in your SQLite database. Here's an example:

```
sqlite> select * from foods where name='JujyFruit' and type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

The restriction here works according to the expression `(name='JujyFruit') and (type_id=9)`, which consists of two logical expressions joined by logical `and`. Both of these conditions must be true for any record in `foods` to be included in the result.

The LIKE and GLOB Operators

A particularly useful relational operator is `like`. `like` is similar to equals (`=`) but is used for matching string values against patterns. For example, to select all rows in `foods` whose names begin with the letter *J*, you could do the following:

```
sqlite> select id, name from foods where name like 'J%';

id     name
-----  --------------------
156    Juice box
236    Juicy Fruit Gum
243    Jello with Bananas
244    JujyFruit
245    Junior Mints
370    Jambalaya
```

A percent symbol (`%`) in the pattern matches any sequence of zero or more characters in the string. An underscore (`_`) in the pattern matches any single character in the string. The percent symbol is greedy. It will eat everything between two characters except those characters. If it is on the extreme left or right of a pattern, it will consume everything on each respective side. Consider the following examples:

```
sqlite> select id, name from foods where name like '%ac%P%';

id     name
-----  -------------------
127    Guacamole Dip
168    Peach Schnapps
198    Mackinaw Peaches
```

Another useful trick is to use `NOT` to negate a pattern:

```
sqlite> select id, name from foods
        where name like '%ac%P%' and name not like '%Sch%'

id     name
-----  -------------------
 38    Pie (Blackberry) Pie
127    Guacamole Dip
198    Mackinaw peaches
```

The `glob` operator is very similar in behavior to the `like` operator. Its key difference is that it behaves much like Unix or Linux file globbing semantics in those operating systems. This means it uses the wildcards associated with file globbing such as * and _ and that matching is case sensitive. The following example demonstrates `glob` in action:

```
sqlite> select id, name from foods
   ...> where name glob 'Pine*';

id          name
----------  ----------
205         Pineapple
258         Pineapple
```

SQLite also recognizes the `match` and `regexp` predicates but currently does not provide a native implementation. You'll need to develop your own with the `sqlite_create_funtion()` call, discussed in Chapters 5 through 8.

Limiting and Ordering

You can limit the size and particular range of the result using the `limit` and `offset` keywords. `limit` specifies the maximum number of records to return. `offset` specifies the number of records to skip. For example, the following statement obtains the Cereal record (the second record in `food_types`) using `limit` and `offset`:

```
select * from food_types order by id limit 1 offset 1;
```

The `offset` clause skips one row (the `Bakery` row), and the `limit` clause returns a maximum of one row (the Cereal row).

But there is something else here as well: `order by`. This clause sorts the result by a column or columns before it is returned. The reason it is important in this example is because the rows returned from `select` are never guaranteed to be in any specific order—the SQL standard declares this. Thus, the `order by` clause is essential if you need to count on the result being in any specific order. The syntax of

the `order by` clause is similar to the `select` clause: it is a comma-separated list of columns. Each entry may be qualified with a sort order—`asc` (ascending, the default) or `desc` (descending). For example:

```
sqlite> select * from foods where name like 'B%'
        order by type_id desc, name limit 10;

id     type_id   name
-----  --------  --------------------
382    15        Baked Beans
383    15        Baked Potato w/Sour
384    15        Big Salad
385    15        Broccoli
362    14        Bouillabaisse
328    12        BLT
327    12        Bacon Club (no turke
326    12        Bologna
329    12        Brisket Sandwich
274    10        Bacon
```

Typically you only need to order by a second (third, and so on) column when there are duplicate values in the first (second, and so on) ordered column(s). Here, there were many duplicate `type_id`s. I wanted to group them together and then arrange the foods alphabetically within these groups.

■ **Note** `limit` and `offset` are not standard SQL keywords as defined in the ANSI standard. Most databases have functional equivalents, although they use different syntax.

If you use both `limit` and `offset` together, you can use a comma notation in place of the `offset` keyword. For example, the following SQL:

```
select * from foods where name like 'B%'
order by type_id desc, name limit 1 offset 2;
```

can be expressed equivalently with this:

```
sqlite> select * from foods where name like 'B%'
        order by type_id desc, name limit 2,1;

id     type_id   name
-----  --------  --------------------
384    15        Big Salad
```

There is a trick here for those who think this syntax should be the other way around. In SQLite, when using this shorthand notation, the `offset` precedes the `limit`. Here, the values following `limit` keyword specify an offset of `2` and a `limit` of 2. Also, note that `offset` depends on limit. That is, you can use limit without using offset but not the other way around.

Notice that `limit` and `offset` are dead last in the operational pipeline. One common misconception of `limit`/`offset` is that it speeds up a query by limiting the number of rows that must be collected by the

`where` clause. This is not true. If it were, then `order by` would not work properly. For `order by` to do its job, it must have the entire result in hand to provide the correct order. There is a small performance boost, in that SQLite only needs to keep track of the order of the 10 biggest values at any point. This has some benefit but not nearly as much as the "sort-free limiting" that some expect.

Functions and Aggregates

SQLite comes with various built-in functions and aggregates that can be used within various clauses. Function types range from mathematical functions such as `abs()`, which computes the absolute value, to string-formatting functions such as `upper()` and `lower()`, which convert text to upper- and lowercase, respectively. For example:

```
sqlite> select upper('hello newman'), length('hello newman'), abs(-12);

upper('hello newman')  length('hello newman') abs(-12)
---------------------  --------------------   ----------
HELLO NEWMAN           12                     12
```

Notice that the function names are case insensitive (i.e., `upper()` and `UPPER()` refer to the same function). Functions can accept column values as their arguments:

```
sqlite> select id, upper(name), length(name) from foods
        where type_id=1 limit 10;

id     upper(name)               length(name)
-----  ------------------------  ------------
1      BAGELS                    6
2      BAGELS, RAISIN            14
3      BAVARIAN CREAM PIE        18
4      BEAR CLAWS                10
5      BLACK AND WHITE COOKIES   23
6      BREAD (WITH NUTS)         17
7      BUTTERFINGERS             13
8      CARROT CAKE               11
9      CHIPS AHOY COOKIES        18
10     CHOCOLATE BOBKA           15
```

Since functions can be part of any expression, they can also be used in the `where` clause:

```
sqlite> select id, upper(name), length(name) from foods
        where length(name) < 5 limit 5;

id     upper(name)           length(name)
-----  --------------------  --------------------
36     PIE                   3
48     BRAN                  4
56     KIX                   3
57     LIFE                  4
80     DUCK                  4
```

Aggregates are a special class of functions that calculate a composite (or aggregate) value over a group of rows (or relation). Standard aggregate functions include `sum()`, `avg()`, `count()`, `min()`, and `max()`. For example, to get a count of the number of foods that are baked goods (`type_id=1`), you can use the `count` aggregate as follows:

```
sqlite> select count(*) from foods where type_id=1;

count
-----
47
```

The `count` aggregate returns a count of every row in the relation. Whenever you see an aggregate, you should automatically think, "For each row in a table, do something."

Aggregates can aggregate not only column values but any expression—including functions. For example, to get the average length of all food names, you can apply the `avg` aggregate to the `length(name)` expression as follows:

```
sqlite> select avg(length(name)) from foods;

avg(length(name))
-----------------
12.58
```

Aggregates operate within the `select` clause. They compute their values on the rows selected by the `where` clause—not from all rows selected by the `from` clause. The `select` command filters first and then aggregates.

Although SQLite comes with a standard set of common SQL functions and aggregates, it is worth noting that the SQLite C API allows you to create custom functions and aggregates as well. See Chapter 7 for more information.

Grouping

An essential part of aggregation is grouping. That is, in addition to computing aggregates over an entire result, you can also split that result into groups of rows with like values and compute aggregates on each group—all in one step. This is the job of the `group by` clause. Here's an example:

```
sqlite> select type_id from foods group by type_id;

type_id
----------
1
2
3
.
.
.
15
```

`group by` is a bit different from the other parts of `select`, so you need to use your imagination a little to wrap your head around it. Figure 3-7 illustrates the process. Operationally, `group by` sits in between the `where` clause and the `select` clause. `group by` takes the output of `where` and splits it into groups of rows that share a common value (or values) for a specific column (or columns). These groups are then passed to the `select` clause. In the example, there are 15 different food types (`type_id` ranges from 1 to 15), and therefore `group by` organizes all rows in foods into 15 groups varying by `type_id`. `select` takes each group, extracts its common `type_id` value, and puts it into a separate row. Thus, there are 15 rows in the result, one for each group.

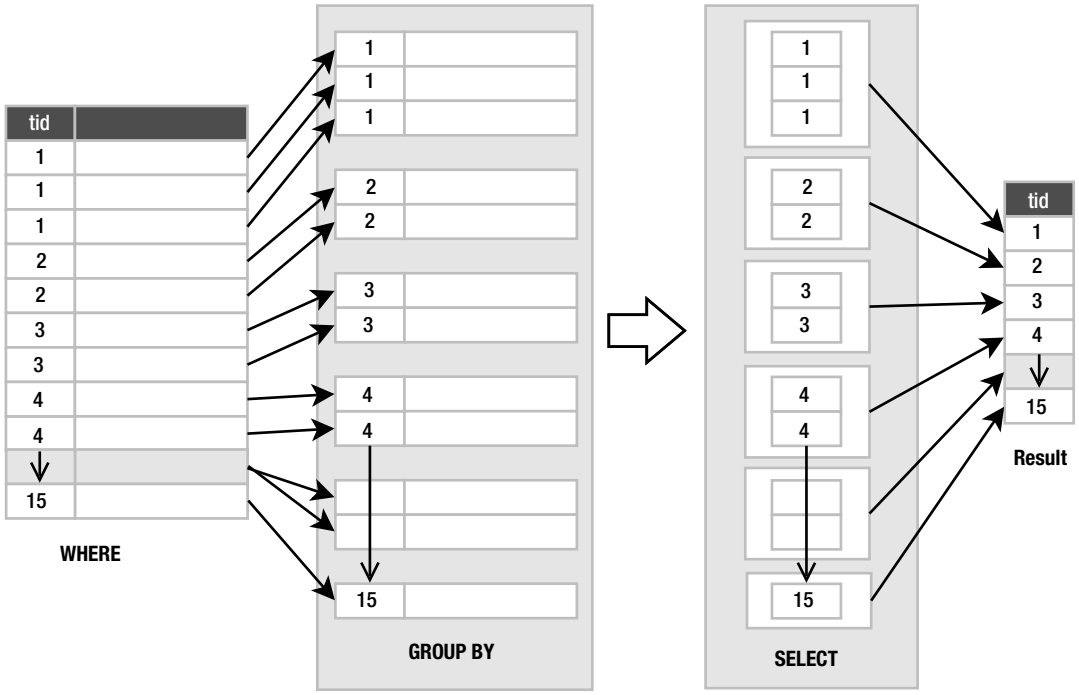


Figure 3-7. `group by` process

When `group by` is used, the `select` clause applies aggregates to each group separately, rather than the entire result as a whole. Since aggregates produce a single value from a group of values, they collapse these groups of rows into single rows. For example, consider applying the `count` aggregate to the preceding example to get the number of records in each `type_id` group:

```
sqlite> select type_id, count(*) from foods group by type_id;

type_id     count(*)
----------  ----------
1           47
2           15
3           23
```

```
4           22
5           17
6           4
7           60
8           23
9           61
10          36
11          16
12          23
13          14
14          19
15          32
```

Here, `count()` was applied 15 times—once for each group, as illustrated in Figure 3-8. Note that in the diagram, the actual number of records in each group is not represented literally (that is, it doesn't show 47 records in the group for `type_id`=1).

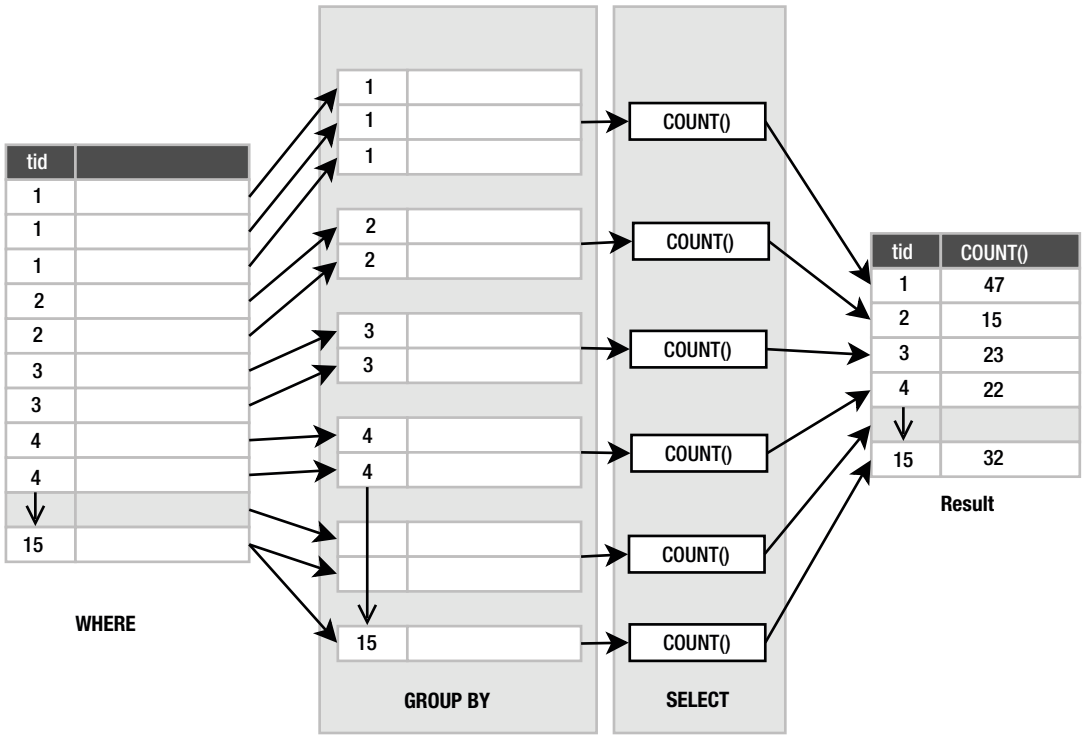


Figure 3-8. *`group by` and aggregation*

The number of records with `type_id=1` (Baked Goods) is 47. The number with `type_id=2` (Cereal) is 15. The number with `type_id=3` (Chicken/Fowl) is 23, and so forth. So, to get this information, you could run 15 queries as follows:

```
select count(*) from foods where type_id=1;
select count(*) from foods where type_id=2;
select count(*) from foods where type_id=3;
.
.
.
select count(*) from foods where type_id=15;
```

Or, you get the results using the single `select` with a `group by` as follows:

```
select type_id, count(*) from foods group by type_id;
```

But there is more. Since `group by` has to do all this work to create groups with like values, it seems a pity not to let you filter these groups before handing them off to the `select` clause. That is the purpose of `having`, a predicate that you apply to the result of `group by`. It filters the groups from `group by` in the same way that the `where` clause filters rows from the `from` clause. The only difference is that the `where` clause's predicate is expressed in terms of individual row values, and `having`'s predicate is expressed in terms of aggregate values.

Take the previous example, but this time say you are interested only in looking at the food groups that have fewer than 20 foods in them:

```
sqlite> select type_id, count(*) from foods
        group by type_id having count(*) < 20;

type_id     count(*)
----------  ----------
2           15
5           17
6           4
11          16
13          14
14          19
```

Download from Wow! eBook <www.wowebook.com>

Here, `having` applies the predicate `count(*)<20` to all the groups. Any group that does not satisfy this condition (that has 20 or more foods in it) is not passed on to the `select` clause. Figure 3-9 illustrates this restriction.

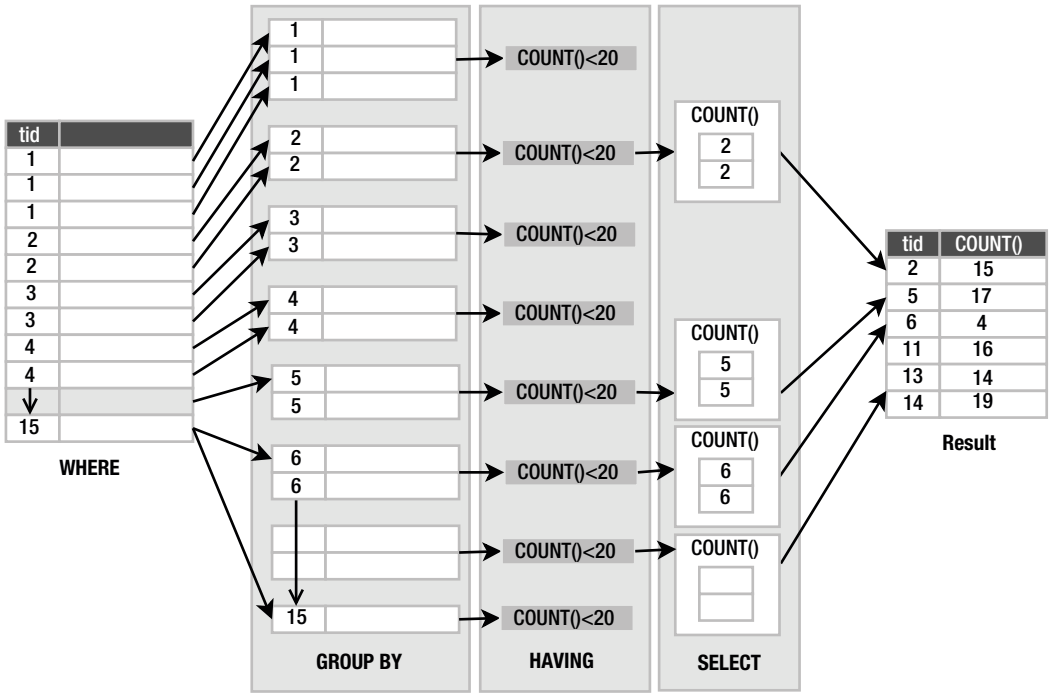


Figure 3-9. `having` as group restriction

The third column in the figure shows groups of rows ordered by `type_id`. The values shown are the type_id values. The actual number of rows shown in each group is not exact but figurative. I have only shown two rows in each group to represent the group as a whole.

So, `group by` and `having` work as additional restriction phases. `group by` takes the restriction produced by the `where` clause and breaks it into groups of rows that share a common value for a given column. `having` then applies a filter to each of these groups. The groups that make it through are passed on to the `select` clause for aggregation and projection.

■ **Caution** Some databases, including SQLite, will allow you to construct a `select` statement where nonaggregated columns are not grouped in a `select` statement with mixed aggregate and nonaggregate columns. For instance, SQLite will allow you to execute this SQL:

```
select type_id, count(*) from foods
```

Because the aggregate (`count`) collapses the input table, there is a mismatch in the number of rows that pass all the filtering steps. Count in this case will return one row, but we haven't instructed SQLite how to group `type_id`. Nevertheless, SQLite will return something. Sadly, the results of such a statement are meaningless—*do not rely* on any SQL statement that doesn't group by nonaggregate fields. *The results are arbitrary!*

Removing Duplicates

`distinct` takes the result of the `select` clause and filters out duplicate rows. For example, you'd use this to get all distinct `type_id` values from `foods`:

```
sqlite> select distinct type_id from foods;

type_id
----------
1
2
3
.
.
.
15
```

This statement works as follows: the `where` clause returns the entire foods table (all 412 records). The `select` clause pulls out just the `type_id` column, and finally `distinct` removes duplicate rows, reducing the number from 412 rows to 15 rows, all unique.

Joining Tables

Joins are the key to working with data from multiple tables (or relations) and are the first operation(s) of the `select` command. The result of a join is provided as the input or starting point for all subsequent (filtering) operations in the `select` command.

Joins in SQLite are best understood by example. The `foods` table has a column `type_id`. As it turns out, the values in this column correspond to values in the `id` column in the `food_types` table. A relationship exists between the two tables. Any value in the `foods.type_id` column must correspond to a value in the `food_types.id` column, and the `id` column is the *primary key* (described later) of `food_types`. The `foods.type_id` column, by virtue of this relationship, is called a *foreign key*: it contains (or references) values in the primary key of another table. This relationship is called a *foreign key relationship*

Using this relationship, it is possible to join the foods and food_type tables on these two columns to make a new relation, which provides more detailed information, namely, the `food_types.name` for each food in the `foods` table. This is done with the following SQL:

```
sqlite> select foods.name, food_types.name
        from foods, food_types
        where foods.type_id=food_types.id limit 10;

name                        name
--------------------------  ---------------
Bagels                      Bakery
Bagels, raisin              Bakery
Bavarian Cream Pie          Bakery
Bear Claws                  Bakery
Black and White cookies     Bakery
Bread (with nuts)           Bakery
Butterfingers               Bakery
Carrot Cake                 Bakery
Chips Ahoy Cookies          Bakery
Chocolate Bobka             Bakery
```

You can see the `foods.name` in the first column of the result, followed by the `food_types.name` in the second. Each row in the former is linked to its associated row in the latter using the `foods.type_id` → `food_types.id` relationship (Figure 3-10).

■ **Note** We are using a new notation in this example to specify the columns in the `select` clause. Rather than specifying just the column names, we are using the notation `table_name.column_name`. The reason is that we have multiple tables in the `select` statement. The database is smart enough to figure out which table a column belongs to—as long as that column name is unique among all tables. If you use a column whose name is also defined in other tables of the join, the database will not be able to figure out which of the columns you are referring to and will return an error. In practice, when you are joining tables, it is always a good idea to use the `table_name.column_name` notation to avoid any possible ambiguity. This is explained in detail in the section "Names and Aliases."

To carry out the join, the database finds these matched rows. For each row in the first table, the database finds all rows in the second table that have the same value for the joined columns and includes them in the input relation. So in this example, the `from` clause built a composite relation by joining the rows of two tables.

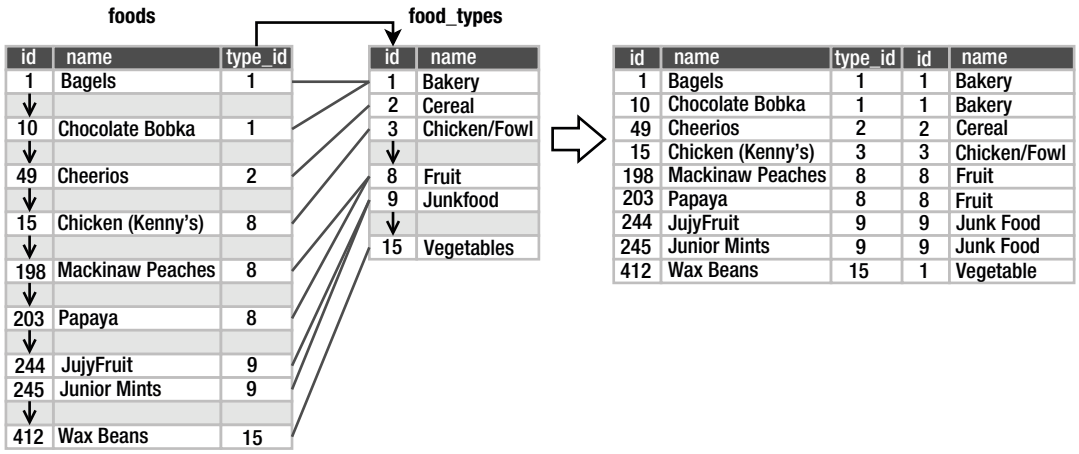


Figure 3-10. *foods and food_types join*

The subsequent operations (`where`, `group by`, and so on) work exactly the same. It is only the input that has changed through joining tables. As it turns out, SQLite supports six different kinds of joins. The one just described, called an *inner join,* is the most common.

Inner Joins

An inner join is where two tables are joined by a relationship between two columns in the tables, as in this previous example. It is the most common (and perhaps most generally useful) type of join.

An inner join uses another set operation in relational algebra, called an *intersection*, to find elements that exist in both sets. Figure 3-11 illustrates this. The intersection of the set {1, 2, 8, 9} and the set {1, 3, 5, 8} is the set {1, 8}. The intersection operation is represented by a Venn diagram showing the common elements of both sets.

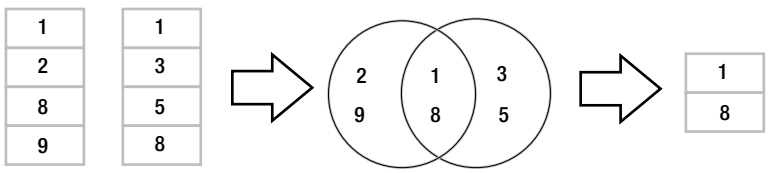


Figure 3-11. Set intersection

This is precisely how an inner join works, but the sets in a join are common elements of the related columns. Pretend that the left set in Figure 3-11 represents values in the `foods.type_id` column and the right set represents values of the `food_types.id` column. Given the matching columns, an inner join finds the rows from both sides that contain like values and combines them to form the rows of the result (Figure 3-12). Note that this example assumes that the records shown are the only records in `foods` and `food_types`.

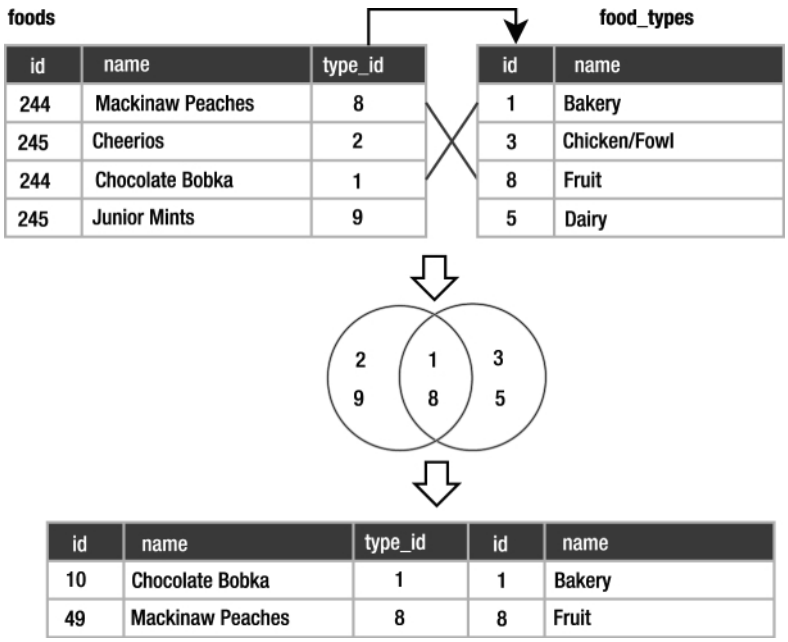


Figure 3-12. *Food join as intersection*

Inner joins only return rows that satisfy the given column relationship, also called the *join condition*. They answer the question, "What rows of B match rows in A given the following relationship?" Let's demonstrate this in SQL so you can match the theory with a practical example.

```
Select *
From foods inner join food_types on foods.id = food_types.id
```

The syntax is clearly understandable once you've visualized what's happening.

Cross Joins

Imagine for a moment that there is no join condition. What would you get? In this case, if the two tables were not related in any way, `select` would produce a more fundamental kind of join (the *most* fundamental), which is called a *cross join*, *Cartesian join*, or *cross product*. The Cartesian join is one of the fundamental relational operations. It is a brute-force, almost nonsensical join that results in the combination of all rows from the first table with all rows in the second.

In SQL, the cross join of `foods` and `food_types` is expressed as follows:

```
select * from foods, food_types;
```

`from`, in the absence of anything else, produces a cross join. The result is shown here. Every row in `foods` is combined with every row in `food_types`, but crucially, not by relating the `id` and `type_id` values. In a cross join, no relationship exists between the rows; there is no join condition, but they are simply jammed together.

```
1           1           Bagels      1           Bakery
1           1           Bagels      2           Cereal
1           1           Bagels      3           Chicken/Fowl
…
(6174 rows omitted to save paper)
…
412         15          Wax Beans   13          Seafood
412         15          Wax Beans   14          Soup
412         15          Wax Beans   15          Vegetables
```

You may ask yourself what purpose such a cross join has. To be frank, that's the question you should always ask yourself before using such logic. Do you really want the cross-product of every row in one table with every row in another. The answer is almost always no.

Outer Joins

Three of the remaining four joins are called outer joins. An inner join selects rows across tables according to a given relationship. An outer join selects all the rows of an inner join plus some rows outside of the relationship. The three outer join types are called *left*, *right*, and *full*. A left outer join operates with respect to the "left table" in the SQL command. For example, in the command:

```
select *
from foods left outer join foods_episodes on foods.id=foods_episodes.food_id;
```

`foods` is the left table here. The left outer join favors it. It is the table of significance in a left outer join. The left outer join tries to match every row of `foods` with every row in `foods_episodes` per the join condition (`foods.id=foods_episodes.food_id`). All such matching rows are included in the result. However, if we had registered some foodstuffs in the `foods` table that had not yet appeared in an episode, the remaining food rows of `foods` that don't match `foods_episodes` are still included in the result, and where `foods_episodes` hasn't provide a row, it "supplies" `null` results.

The right outer join works similarly, except the right table is the one whose rows are included, matching or not. Operationally, left and right outer joins are identical; they do the same thing. They differ only in order and syntax. As a user of SQLite, that means any problem that can requires a right outer join for solution can equally be solved with a left outer join.

A full outer join is the combination of a left and right outer join. It includes all matching records, followed by unmatched records in the right and left tables. Currently, both right and full outer joins are not supported in SQLite. However, as mentioned earlier, a right join can be replaced with a left outer join, and a full outer join can be performed using compound queries (see the section "Compound Queries" later in this chapter).

Natural Joins

The last join on the list is called a *natural join*. It is actually an inner join in disguise, but with a little syntax and convention thrown in. A natural join joins two tables by their common column names. Thus, using the natural join, you can get the inner join of two tables without having to add the join condition.

Natural joins will join *all* columns by the same name in both tables. Just the process of adding to or removing a column from a table can drastically change the results of a natural join query. This can produce very unpredictable results, especially if your tables' design change over time. It's always better to explicitly define the join conditions of your queries than rely on the semantics of the table schema.

Preferred Syntax

Syntactically, there are various ways of specifying a join. The inner join example of `foods` and `food_types` illustrates performing a join implicitly in the `where` clause:

```
select * from foods, food_types where foods.id=food_types.food_id;
```

When the database sees more than one table listed, it knows there will be a join—at the very least a cross join. The `where` clause here calls for an inner join.

This implicit form, although rather clean, is actually an older form of syntax that you should avoid. The politically correct way (per SQL92) to express a join in SQL is using the join keyword. The general form is as follows:

```
select heading from left_table join_type right_table on join_condition;
```

This explicit form can be used for all join types. For example:

```
select * from foods inner join food_types on foods.id=food_types.food_id;
select * from foods left outer join food_types on foods.id=food_types.food_id;
select * from foods cross join food_types;
```

The most important reason for using ANSI join syntax is that there are some query types that can only be satisfied by using the join keyword-style syntax. This is particularly so for any form of outer join—left, right, or full.

Names and Aliases

When joining tables, ambiguity can arise if both tables have a column with the same name. If you were to join two tables with and `id` column using a `select id` clause in the earlier code, which `id` should SQLite return? To help with this type of task, you can qualify column names with their table names to remove any ambiguity, as you saw earlier.

Another useful feature is *aliases*. If your table name is particularly long and you don't want to have to use its name every time you qualify a column, you can use an alias. Aliasing is actually a fundamental relational operation called *rename*. The rename operation simply assigns a new name to a relation. For example, consider this statement:

```
select foods.name, food_types.name
from foods, food_types
where foods.type_id = food_types.id
limit 10;
```

There is a good bit of typing here. You can alias the tables in the `source` clause by simply including the new name directly after the table name, as in the following example:

```
select f.name, t.name
from foods f, food_types t
where f.type_id = t.id
limit 10;
```

Here, the `foods` table is assigned the alias `f`, and the `food_types` table is assigned the alias `t`. Now, every other reference to `foods` or `food_types` in the statement must use the alias `f` and `t`, respectively. Aliases make it possible to do *self-joins*—joining a table with itself. For example, say you want to know what foods in season 4 are mentioned in other seasons. You would first need to get a list of episodes and foods in season 4, which you would obtain by joining `episodes` and `episodes_foods`. But then you would need a similar list for foods outside of season 4. Finally, you would combine the two lists based on their common foods. The following query uses self-joins to do the trick:

```
select f.name as food, e1.name, e1.season, e2.name, e2.season
from episodes e1, foods_episodes fe1, foods f,
     episodes e2, foods_episodes fe2
where
  -- Get foods in season 4
  (e1.id = fe1.episode_id and e1.season = 4) and fe1.food_id = f.id
  -- Link foods with all other episodes
  and (fe1.food_id = fe2.food_id)
  -- Link with their respective episodes and filter out e1's season
  and (fe2.episode_id = e2.id AND e2.season != e1.season)
order by f.name;
```

```
food                 name               season  name                      season
-------------------  -----------------  ------  ------------------------  ------
Bouillabaisse        The Shoes          4       The Stake Out             1
Decaf Cappuccino     The Pitch          4       The Good Samaritan        3
Decaf Cappuccino     The Ticket         4       The Good Samaritan        3
Egg Salad            The Trip 1         4       Male Unbonding            1
Egg Salad            The Trip 1         4       The Stock Tip             1
Mints                The Trip 1         4       The Cartoon               9
Snapple              The Virgin         4       The Abstinence            8
Tic Tacs             The Trip 1         4       The Merv Griffin Show     9
Tic Tacs             The Contest        4       The Merv Griffin Show     9
Tuna                 The Trip 1         4       The Stall                 5
Turkey Club          The Bubble Boy     4       The Soup                  6
Turkey Club          The Bubble Boy     4       The Wizard                9
```

I have put comments in the SQL to better explain what is going on. This example uses two self-joins using the `where` clause join syntax. There are two instances of `episodes` and `foods_episodes`, but they are treated as if they are two independent tables. The query joins foods_episodes back on itself to link the

two instances of `episodes`. The two `episodes` instances are related to each other by an inequality condition to ensure that they are in different seasons. You can alias column names and expressions in the same way. The general alias syntax in SQLite is the same in all cases.

```
Select base-name [[as] alias] …
```

The `as` keyword is optional, but many people prefer it because it makes the aliasing more legible, and it makes you less likely to confuse an alias with a base column name or expression.

Subqueries

Subqueries are `select` statements within `select` statements. They are also called *subselects*. Subqueries are useful in many ways, they work anywhere normal expressions work, and they can therefore be used in a variety of places in a `select` statement. Subqueries are useful in other commands as well.

Perhaps the most common use of subqueries is in the `where` clause, specifically using the `in` operator. The `in` operator is a binary operator that takes an input value and a list of values and returns true if the input value exists in the list, or false otherwise. Here's an example:

```
sqlite> select 1 in (1,2,3);
1
sqlite> select 2 in (3,4,5);
0
sqlite> select count(*)
   ...> from foods
   ...> where type_id in (1,2);
62
```

Using a subquery, you can rewrite the last statement in terms of names from the `food_types`:

```
sqlite> select count(*)
   ...> from foods
   ...> where type_id in
   ...>  (select id
   ...>   from food_types
   ...>   where name='Bakery' or name='Cereal');
62
```

Subqueries in the `select` clause can be used to add additional data from other tables to the result set. For example, to get the number of episodes each food appears in, the actual count from `foods_episodes` can be performed in a subquery in the `select` clause:

```
sqlite> select name,
        (select count(id) from foods_episodes where food_id=f.id) count
        from foods f order by count desc limit 10;

name            count
----------      ----------
Hot Dog         5
Pizza           4
Ketchup         4
```

```
Kasha           4
Shrimp          3
Lobster         3
Turkey Sandwich 3
Turkey Club     3
Egg Salad       3
Tic Tacs        3
```

The `order by` and `limit` clauses here serve to create a top ten list, with hot dogs at the top. Notice that the subquery's predicate references a table in the enclosing `select` command: `food_id=f.id`. The variable `f.id` exists in the outer query. The subquery in this example is called a *correlated subquery* because it references, or correlates to, a variable in the outer (enclosing) query.

Subqueries can be used in the `order by` clause as well. The following SQL groups foods by the size of their respective food groups, from greatest to least:

```
select * from foods f
order by (select count(type_id)
from foods where type_id=f.type_id) desc;
```

`order by` in this case does not refer to any specific column in the result. How does this work then? The `order by` subquery is run for each row, and the result is associated with the given row. You can think of it as an invisible column in the result set, which is used to order rows.

Finally, we have the `from` clause. There may be times when you want to join to the results of another query, rather than to a base table. It's yet another job for a subquery:

```
select f.name, types.name from foods f
inner join (select * from food_types where id=6) types
on f.type_id=types.id;
```

```
name                      name
------------------------  -----
Generic (as a meal)       Dip
Good Dip                  Dip
Guacamole Dip             Dip
Hummus                    Dip
```

Notice that the use of a subquery in the `from` clause requires a rename operation. In this case, the subquery was named `types`. Subqueries as a source relation in the `from` clause are often called *inline views* or *derived tables*.

The thing to remember about subqueries is that they can be used *anywhere* a relational expression can be used. A good way to learn how, where, and when to use them is to just play around with them and see what you can get away with. There is often more than one way to skin a cat in SQL. When you understand the big picture, you can make more informed decisions on when a query might be rewritten to run more efficiently.

Download from Wow! eBook <www.wowebook.com>

Compound Queries

Compound queries are kind of the inverse of subqueries. A compound query is a query that processes the results of multiple queries using three specific relational operations: union, intersection, and difference. In SQLite, these are defined using the `union`, `intersect`, and `except` keywords, respectively.

Compound query operations require a few things of their arguments:

- The relations involved must have the same number of columns.
- There can be only one `order by` clause, which is at the end of the compound query and applies to the combined result.

Furthermore, relations in compound queries are processed from left to right.

The `union` operation takes two relations, *A* and *B*, and combines them into a single relation containing all distinct rows of *A* and *B*. In SQL, `union` combines the results of two `select` statements. By default, `union` eliminates duplicates. If you want duplicates included in the result, then use `union all`. For example, the following SQL finds the single most and single least frequently mentioned foods:

```
select f.*, top_foods.count from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) desc limit 1) top_foods
  on f.id=top_foods.food_id
union
select f.*, bottom_foods.count from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) limit 1) bottom_foods
  on f.id=bottom_foods.food_id
order by top_foods.count desc;
```

```
id     type_id  name       top_foods.count
-----  -------  ---------  ---------------
288    10       Hot Dog    5
1      1        Bagels     1
```

Both queries return only one row. The only difference in the two is which way they sort their results. The `union` simply combines the two rows into a single relation.

The `intersect` operation takes two relations, A and B, and selects all rows in A that also exist in B. The following SQL uses `intersect` to find the all-time top ten foods that appear in seasons 3 through 5:

```
select f.* from foods f
inner join
  (select food_id, count(food_id) as count
     from foods_episodes
     group by food_id
     order by count(food_id) desc limit 10) top_foods
  on f.id=top_foods.food_id
```

```
intersect
select f.* from foods f
  inner join foods_episodes fe on f.id = fe.food_id
  inner join episodes e on fe.episode_id = e.id
  where e.season between 3 and 5
order by f.name;
```

```
id     type_id  name
-----  -------  --------------------
4      1        Bear Claws
146    7        Decaf Cappuccino
153    7        Hennigen's
55     2        Kasha
94     4        Ketchup
164    7        Naya Water
317    11       Pizza
```

To produce the top ten foods, we needed an `order by` in the first `select` statement. Since compound queries allow only one `order by` at the end of the statement, we got around this by performing an inner join on a subquery in which we computed the top ten most common foods. Subqueries can have `order by` clauses because they run independently of the compound query. The inner join then produces a relation containing the top ten foods. The second query returns a relation containing all foods in episodes 3 through 5. The `intersect` operation then finds all matching rows.

The `except` operation takes two relations, A and B, and finds all rows in A that are not in B. By changing the `intersect` to `except` in the previous example, you can find which top ten foods are not in seasons 3 through 5:

```
select f.* from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) desc limit 10) top_foods
  on f.id=top_foods.food_id
except
select f.* from foods f
  inner join foods_episodes fe on f.id = fe.food_id
  inner join episodes e on fe.episode_id = e.id
  where e.season between 3 and 5
order by f.name;
```

```
id     type_id  name
-----  -------  --------
192    8        Banana
133    7        Bosco
288    10       Hot Dog
```

As mentioned earlier, what is called the `except` operation in SQL is referred to as the *difference* operation in relational algebra.

Compound queries are useful when you need to process similar data sets that are materialized in different ways. Basically, if you cannot express everything you want in a single `select` statement, you can use a compound query to get part of what you want in one `select` statement and part in another (and perhaps more) and process the sets accordingly.

Conditional Results

The `case` expression allows you to handle various conditions within a `select` statement. There are two forms. The first and simplest form takes a static value and lists various case values linked to return values:

```
case value
  when x then value_x
  when y then value_y
  when z then value_z
  else default_value
end
```

Here's a simple example:

```
select name || case type_id
                 when 7  then ' is a drink'
                 when 8  then ' is a fruit'
                 when 9  then ' is junkfood'
                 when 13 then ' is seafood'
                 else null
               end description
from foods
where description is not null
order by name
limit 10;
```

```
description
--------------------------------------------
All Day Sucker is junkfood
Almond Joy is junkfood
Apple is a fruit
Apple Cider is a drink
Apple Pie is a fruit
Arabian Mocha Java (beans) is a drink
Avocado is a fruit
Banana is a fruit
Beaujolais is a drink
Beer is a drink
```

The `case` expression in this example handles a few different `type_id` values, returning a string appropriate for each one. The returned value is called `description`, as qualified after the `end` keyword. This string is concatenated to `name` by the string concatenation operator (`||`), making a complete

sentence. For all type_ids not specified in a when condition, case returns null. The select statement filters out such null values in the where clause, so all that is returned are rows that the case expression does handle.

The second form of case allows for expressions in the when condition. It has the following form:

```
case
  when condition1 then value1
  when condition2 then value2
  when condition3 then value3
  else default_value
end
```

CASE works equally well in subselects comparing aggregates. The following SQL picks out frequently mentioned foods:

```
select name,(select
               case
                 when count(*) > 4 then 'Very High'
                 when count(*) = 4 then 'High'
                 when count(*) in (2,3) then 'Moderate'
                 else 'Low'
               end
             from foods_episodes
             where food_id=f.id) frequency
from foods f
where frequency like '%High'
```

```
name       frequency
---------  ----------
Kasha      High
Ketchup    High
Hot Dog    Very High
Pizza      High
```

This query runs a subquery for each row in foods that classifies the food by the number of episodes it appears in. The result of this subquery is included as a column called frequency. The where predicate filters frequency values that have the word *High* in them.

Only one condition is executed in a case expression. If more than one condition is satisfied, only the first of them is executed. If no conditions are satisfied and no else condition is defined, case returns null.

Handling Null in SQLite

Most relational databases support the concept of "unknown" or "unknowable" through a special placeholder called null, which is a placeholder for missing information and is not a value per se. Rather, null is the absence of a value: null is not nothing, null is not something, null is not true, null is not false, null is not zero, null is not an empty string. Simply put, null is resolutely what it is: null. And not

everyone can agree on what that means. There are a few key rules and ideas that you should learn so you can master the use of `null` in SQLite.

First, in order to accommodate `null` in logical expressions, SQL uses something called *three-value* (or *tristate*) logic, where `null` is one of the truth values. Table 3-2 shows the truth table for logical `and` and logical `or` with `null` thrown into the mix.

Table 3-2. *`AND` and `OR` with `NULL`*

x	y	x AND y	y OR y
True	True	True	True
True	False	False	True
True	`NULL`	`NULL`	True
False	False	False	False
False	`NULL`	False	`NULL`
`NULL`	`NULL`	`NULL`	`NULL`

You can try a few simple select statements to see the effect of `null` yourself.

Second, detecting the presence or absence of `null` is done using the `is null` or `is not null` operator. If you try to use any other operator, such as equals, greater than, and so on, you will receive a shock, which leads us to our third rule regarding `null`.

Third and last, always remember that `null` is not equal to any other value, even `null` itself. You cannot compare `null` with a value, and no `null` is greater than, smaller than, or in any way related to any another `null`. This often catches out the unwary when statements like the next example fail to return any rows.

```
select *
from mytable
where myvalue = null;
```

Nothing can equal `null`, so no data will ever be returned by that statement in SQLite. Now that you've mastered the basics, you'll be pleased to know that SQLite offers several additional functions for working with `null`. The first of which is a workaround for the very limitation of "nothing equals `NULL`" that we've just drilled in to you. From SQLite v 3.6.19, the `is` operator can be used to equate one `NULL` to another. In the most basic form, you can run a simple query to ask SQLite if `NULL is NULL`:

```
sqlite> select NULL is NULL;
1
```

As already mentioned, any value other than zero means "true," so SQLite is happy to tell us that in this instance `NULL` really is the same as `NULL`. But please, don't rely on this heavily. SQL's trivalue logic may be awkward, but it is a standard, and working around it with the `is` operator will likely have you encountering problems with other systems and programming languages that expect the standard behavior.

The `coalesce` function, which is part of the SQL99 standard, takes a list of values and returns the first non-`null` in the list. Take the following example:

```
select coalesce(null, 7, null, 4)
```

In this case, `coalesce` will return 7 as the first non-`null` value. This is also handy when performing arithmetic, so you can detect whether null has been returned and return a meaningful value like 0 instead.

Conversely, the `nullif` function takes two arguments and returns `null` if they have the same values; otherwise, it returns the first argument:

```
sqlite> select nullif(1,1);
null
sqlite> select nullif(1,2);
1
```

If you use `null`, you need to take special care in queries that refer to columns that may contain `null` in their predicates and aggregates. `null` can do quite a number on aggregates if you are not careful.

Summary

Congratulations, you have learned the `select` command for SQLite's implementation of SQL. Not only have you learned how the command works, but you've learned some relational theory in the process. You should now be comfortable with using `select` statements to query your data, join, aggregate, summarize, and dissect it for various uses.

We'll continue the discussion of SQL in the next chapter, where we'll build on your knowledge of `select` by introducing the other members of DML, as well as DDL and other helpful SQL constructs in SQLite.

CHAPTER 4

Advanced SQL for SQLite

You mastered the `select` statement in Chapter 3, so now it is time to add the rest of the SQL dialect to your capabilities. This chapter covers the remainder of SQL as implemented in SQLite, including `insert`, `update`, and `delete` statements to modify data; constraints to protect your data; and more advanced topics such as table creation and data types.

Modifying Data

Compared to `select`, the commands used to modify data are quite easy to use and understand. There are three Data Manipulation Language (DML) commands for modifying data—`insert`, `update`, and `delete`—and they do pretty much what their names imply.

Inserting Records

You insert records into a table using the `insert` command. `insert` works on a single table and can both insert one row at a time and can insert many rows at once using a `select` command. The general form of the `insert` command is as follows:

```
insert into table (column_list) values (value_list);
```

The variable table specifies which table—the target table—to insert into. The variable `column_list` is a comma-separated list of column names, all of which must exist in the target table. The variable `value_list` is a comma-separated list of values that correspond to the names given in `column_list`. The order of values in `value_list` must correspond to the order of columns in `column_list`.

Inserting One Row

For example, you'd use this to insert a row into `foods`:

```
sqlite> insert into foods (name, type_id) values ('Cinnamon Bobka', 1);
```

This statement inserts one row, specifying two column values. `'Cinnamon Bobka'`—the first value in the value list—corresponds to the column `name`, which is the first column in the column list. Similarly, the value 1 corresponds to `type_id`, which is listed second. Notice that `id` was not mentioned. In this case,

the database uses the default value. Since `id` is declared as `integer primary key`, it will be automatically generated and associated with the record (as explained in the section "Primary Key Constraints"). The inserted record can be verified with a simple `select`, and the value used for `id` can be queried as well:

```
sqlite> select * from foods where name='Cinnamon Bobka';

id          type_id     name
----------  ----------  --------------
413         1           Cinnamon Bobka

sqlite> select max(id) from foods;

MAX(id)
----------
413

sqlite> select last_insert_rowid();

last_insert_rowid()
-------------------
413
```

Notice that the value 413 was automatically generated for `id`, which is the largest value in the column. Thus, SQLite provided a monotonically increasing value. You can confirm this with the built-in SQL function `last_insert_rowid()`, which returns the last automatically generated key value, as shown in the example.

If you provide a value for every column of a table in `insert`, then the column list can be omitted. In this case, the database assumes that the order of values provided in the value list corresponds to the order of columns as declared in the `create table` statement. Here's an example:

```
sqlite> insert into foods values(NULL, 1, 'Blueberry Bobka');
sqlite> select * from foods where name like '%Bobka';

id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
```

Notice here the order of arguments. `'Blueberry Bobka'` came after 1 in the value list. This is because of the way the table was declared. To view the table's definition, type `.schema foods` at the shell prompt:

```
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

The first column is `id`, followed by `type_id`, followed by `name`. This, therefore, is the order you must list values in `insert` statements on foods. Why did the preceding `insert` statement use a `NULL` for `id`? SQLite

knows that `id` in `foods` is an autoincrement column, and specifying a `NULL` is the equivalent of not providing a value at all. Not specifying a value triggers the automatic key generation. It's just a convenient trick. There is no deeper meaning or theoretical basis behind it. We will look at the subtleties of autoincrement columns later in this chapter.

Inserting a Set of Rows

Subqueries can be used in `insert` statements, both as components of the value list and as a complete replacement of the value list. When you specify a subquery as the value list, you are really inserting a set of rows, because you are inserting the set of rows returned by that subquery. Here's an example that generates a set having just one row:

```
insert into foods
values (null,
       (select id from food_types where name='Bakery'),
       'Blackberry Bobka');
select * from foods where name like '%Bobka';
```

```
id          type_id     name
----------  ----------  ----------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobka
```

Here, rather than hard-coding the `type_id` value, I had SQLite look it up for me. Here's another example:

```
insert into foods
select last_insert_rowid()+1, type_id, name from foods
where name='Chocolate Bobka';
select * from foods where name like '%Bobka';
```

```
id          type_id     name
----------  ----------  ----------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobks
416         1           Chocolate Bobka
```

This query completely replaces the value list with a `select` statement. As long as the number of columns in the `select` clause matches the number of columns in the table (or the number of columns in the columns list, if provided), `insert` will work just fine. Here, this example added another chocolate bobka and used the expression `last_insert_rowid()+1` as the `id` value. You could have just as easily used `NULL` instead. In fact, you probably should use `NULL` rather than `last_insert_rowid()`, because `last_insert_rowid()` will return 0 if you did not previously insert a row in the current session.

You could safely assume that this would work properly for these examples, but it would not be a good idea to make this assumption in a program.

Inserting Multiple Rows

There is nothing stopping you from inserting multiple rows at a time using the `select` form of `insert`. As long as the number of columns matches, `insert` will insert every row in the result. Here's an example:

```
sqlite> create table foods2 (id int, type_id int, name text);
sqlite> insert into foods2 select * from foods;
sqlite> select count(*) from foods2;

count(*)
--------------------
418
```

This creates a new table `foods2` and inserts into it all the records from `foods`.

However, there is an easier way to create and populate a table. The `create table` statement has a special syntax for creating tables from `select` statements. The previous example could have been performed in one step using this syntax:

```
sqlite> create table foods2 as select * from foods;
sqlite> select count(*) from list;

count(*)
--------
418
```

`create table` does both steps in one fell swoop. This can be especially useful for creating temporary tables:

```
create temp table list as
select f.name food, t.name name,
       (select count(episode_id)
        from foods_episodes where food_id=f.id) episodes
from foods f, food_types t
where f.type_id=t.id;
select * from list;
```

```
Food                 Name        Episodes
-------------------  ----------  ----------
Bagels               Bakery      1
Bagels, raisin       Bakery      2
Bavarian Cream Pie   Bakery      1
Bear Claws           Bakery      3
Black and White cook Bakery      2
Bread (with nuts)    Bakery      1
Butterfingers        Bakery      1
```

Download from Wow! eBook <www.wowebook.com>

```
Carrot Cake           Bakery      1
Chips Ahoy Cookies    Bakery      1
Chocolate Bobka       Bakery      1
```

When using this form of `CREATE TABLE`, be aware that any constraints defined in the source table are not created in the new table. Specifically, the autoincrement columns will not be created in the new table, nor will indexes, `UNIQUE` constraints, and so forth. Many other databases refer to this approach as *CTAS*, which stands for Create Table As Select, and that phrase is not uncommon among SQLite users.

It is also worth mentioning here that you have to be aware of `unique` constraints when inserting rows. If you add duplicate values on columns that are declared as `unique`, SQLite will stop you in your tracks:

```
sqlite> select max(id) from foods;

max(id)
-------
416

sqlite> insert into foods values (416, 1, 'Chocolate Bobka');
SQL error: PRIMARY KEY must be unique
```

Updating Records

You update records in a table using the `update` command. The `update` command modifies one or more columns within one or more rows in a table. `update` has the following general form:

```
update table set update_list where predicate;
```

`update_list` is a list of one or more column assignments of the form `column_name=value`. The `where` clause works exactly as in `select`. Half of `update` is really a `select` statement. The `where` clause identifies rows to be modified using a predicate. Those rows then have the update list applied to them. Here's an example:

```
update foods set name='CHOCOLATE BOBKA'
where name='Chocolate Bobka';
select * from foods where name like 'CHOCOLATE%';
```

```
id     type_  name
-----  -----  ------------------------------
10     1      CHOCOLATE BOBKA
11     1      Chocolate Eclairs
12     1      Chocolate Cream Pie
222    9      Chocolates, box of
223    9      Chocolate Chip Mint
224    9      Chocolate Covered Cherries
```

`update` is a very simple and direct command, and this is pretty much the extent of its use. As in `insert`, you must be aware of any `unique` constraints, because they will stop `update` every bit as much as `insert`:

```
sqlite> update foods set id=11 where name='CHOCOLATE BOBKA';
SQL error: PRIMARY KEY must be unique
```

This is true for any constraint, however.

Deleting Records

You delete records from a table using the `delete` command. The `delete` command deletes rows from a table. `delete` has the following general form:

```
delete from table where predicate;
```

Syntactically, `delete` is a watered-down `update` statement. Remove the `SET` clause from `update`, and you have `delete`. The `where` clause works exactly like `select`, except that it identifies rows to be deleted. Here's an example:

```
delete from foods where name='CHOCOLATE BOBKA';
```

Data Integrity

Data integrity is concerned with defining and protecting relationships within and between tables. There are four general types: *domain integrity, entity integrity, referential integrity,* and *user-defined integrity.* Domain integrity involves controlling values within columns. Entity integrity involves controlling rows in tables. Referential integrity involves controlling rows between tables—specifically foreign key relationships. And user-defined integrity is a catchall for everything else.

Data integrity is implemented using *constraints.* A constraint is a control measure used to restrict the values that can be stored in a column or columns. Going by just the values in columns, the database can enforce all four types of integrity constraints. In SQLite, constraints also include support for *conflict resolution.* Conflict resolution is covered in detail later in this chapter.

Let's deviate from our `foods` database for a moment and focus on the same `contacts` table introduced in Chapter 3. The `contacts` table is defined as follows:

```
create table contacts (
id integer primary key,
name text not null collate nocase,
phone text not null default 'UNKNOWN',
unique (name,phone) );
```

As you know by now, constraints are part of a table's definition. They can be associated with a column definition or defined independently in the body of the table definition. Column-level constraints include `not null`, `unique`, `primary key`, `foreign key`, `check`, and `collate`. Table-level constraints include `primary key`, `unique`, and `check`. All of these constraints are covered in the following sections according to their respective integrity types.

Now that you have familiarity with the `update`, `insert`, and `delete` commands in SQLite, the operation of many of the constraints will make sense. Just as these commands operate on data, constraints operate on them, making sure that they work within the guidelines defined in the tables they modify.

Entity Integrity

Relational theory—as implemented in most databases including SQLite—requires that every field in the database must be uniquely identifiable and capable of being located. For a field to be addressable, its corresponding row must also be addressable. And for that, the row must be unique in some way. This is the job of the primary key.

The primary key consists of least one column or a group of columns with a `unique` constraint. The `unique` constraint, as you will soon see, simply requires that every value in a column (or group of columns) be distinct. Therefore, the primary key ensures that each row is somehow distinct from all other rows in a table, ultimately ensuring that every field is also addressable. Entity integrity basically keeps data organized in a table. After all, what good is data if you can't find it?

Unique Constraints

Since primary keys are based on `unique` constraints, we'll start with them. A `unique` constraint simply requires that all values in a column or a group of columns are distinct from one another or unique. If you attempt to insert a duplicate value or update a value to another value that already exists in the column, the database will issue a constraint violation and abort the operation. `unique` constraints can be defined at the column or the table level. When defined at the table level, `unique` constraints can be applied across multiple columns. In this case, the combined value of the columns must be unique. In `contacts`, there is a unique constraint on both `name` and `phone` together. See what happens if we attempt to insert another `'Jerry'` record with a `phone` value `'UNKNOWN'`:

```
sqlite> insert into contacts (name,phone) values ('Jerry','UNKNOWN');
SQL error: columns name, phone are not unique

sqlite> insert into contacts (name) values ('Jerry');
SQL error: columns name, phone are not unique

sqlite> insert into contacts (name,phone) values ('Jerry', '555-1212');
```

In the first case, I explicitly specified `name` and `phone`. This matched the values of the existing record, and the `unique` constraint kicked in and did not let me do it. The third `insert` illustrates that the `unique` constraint applies to `name` and `phone` combined, not individually. It inserted another row with `'Jerry'` as the value for `name`, which did not cause an error, because `name` by itself it not unique—only `name` and `phone` together.

NULL AND UNIQUE

Based on your knowledge of `NULL` from Chapter 3 and your new knowledge of `unique` constraints, how many `NULL` values can you put in a column that is declared `unique`? From a theoretical perspective, the answer is as many `NULL`s as you like. Remember that `NULL` is not equal to anything else, even other `NULL`s. And this is the basis for how SQLite handles `NULL` entries in `unique` columns. It's also how other major databases like Oracle and PostgreSQL operate. But the database community is divided on this issue. Users of Informix, Sybase, and Microsoft SQL Server are limited to only one `NULL` entry in a unique column. Those using DB2 are forbidden any at all!

Primary Key Constraints

In SQLite, a primary key column is always created when you create a table, whether you define one or not. This column is a 64-bit integer value called `rowid`. It has two aliases, `_rowid_` and `oid`, which can be used to refer to it as well. Default values are automatically generated for it.

SQLite provides an autoincrement feature for primary keys, should you want to define your own. If you define a column's type as `integer primary key`, SQLite will create a `default` value on that column, which will provide an integer value that is guaranteed to be unique in that column. In reality, however, this column will simply be an alias for `rowid`. They will all refer to the same value. Since SQLite uses a 64-bit signed integer for the primary key, the maximum value for this column is 9,223,372,036,854,775,807.

Even if you manage to reach this limit, SQLite will simply start searching for unique values that are not in the column for subsequent `insert`s. When you delete rows from the table, `rowids` may be recycled and reused on subsequent `insert`s. As a result, newly created `rowids` might not always be in strictly ascending order.

■ **Caution** You will remember our discussion on relational data not having implicit order when we discussed the `order by` statement in Chapter 3. This is another strong reminder never to assume data in a relational database such as SQLite has any particular order—even when your instincts say it should be safe.

If you want SQLite to use unique automatic primary key values for the life of the table—and not "fill in the gaps"—then add the `autoincrement` keyword after the `integer primary key` clause.

In the examples so far, you have managed to insert two records into `contacts`. Not once did you specify a value for `id`. As mentioned before, this is because `id` is declared as `integer primary key`. Therefore, SQLite supplied an integer value for each `insert` automatically, as you can see here:

```
sqlite> select * from contacts;

id   name   phone
---  -----  --------
1    Jerry  UNKNOWN
2    Jerry  555-1212
```

Notice that the primary key is accessible from all the aforementioned aliases, in addition to `id`:

```
sqlite> select rowid, oid,_rowid_,id, name, phone from contacts;

id  id  id  id  name   phone
--  --  --  --  ----   -----
1   1   1   1   Jerry  UNKNOWN
2   2   2   2   Jerry  555-1212
```

If you include the keyword `autoincrement` after `integer primary key`, SQLite will use a different key generation algorithm for the column. This algorithm basically prevents `rowids` from being recycled. It guarantees that only new (not recycled) `rowids` are provided for every `insert`. When a table is created with a column containing the `autoincrement` constraint, SQLite will keep track of that column's maximum `rowid` in a system table called `sqlite_sequence`. It will only use values greater than that maximum on all subsequent `inserts`. If you ever reach the absolute maximum, then SQLite will return a `SQLITE_FULL` error on subsequent `inserts`. Here's an example:

```
sqlite> create table maxed_out(id integer primary key autoincrement, x text);
sqlite> insert into maxed_out values (9223372036854775807, 'last one');
sqlite> select * from sqlite_sequence;

name        seq
----------  -------------------
maxed_out   9223372036854775807

sqlite> insert into maxed_out values (null, 'will not work');
SQL error: database is full
```

Here, we provided the primary key value. SQLite then stored this value as the maximum for `maxed_out.id` in `sqlite_sequence`. We supplied the very last (maximum) 64-bit value before wraparound. In the next `insert`, you used the generated default value, which must be a monotonically increasing value. This wrapped around to 0, and SQLite issued a `SQLITE_FULL` error.

Although SQLite tracks the maximum value for an `autoincrement` column in the `sqlite_sequence` table, it does not prevent you from providing your own values for it in the `insert` command. The only requirement is that the value you provide must be unique within the column. Here's an example:

```
sqlite> drop table maxed_out;
sqlite> create table maxed_out(id integer primary key autoincrement, x text);
sqlite> insert into maxed_out values(10, 'works');
sqlite> select * from sqlite_sequence;
```

```
name        seq
----------  ----------
maxed_out   10

sqlite> insert into maxed_out values(9, 'works');
sqlite> select * from sqlite_sequence;

name        seq
----------  ----------
maxed_out   10

sqlite> insert into maxed_out values (9, 'fails');
SQL error: PRIMARY KEY must be unique

sqlite> insert into maxed_out values (null, 'should be 11');
sqlite> select * from maxed_out;

id          x
----------  ------------
9           works
10          works
11          should be 11

sqlite> select * from sqlite_sequence;

name        seq
----------  ----------
maxed_out   11
```

Here, you dropped and re-created the `maxed_out` table and inserted a record with an explicitly defined `rowid` of 10. Then you inserted a record with a `rowid` less than 10, which worked. You tried it again with the same value and it failed, because of the `unique` constraint. Finally, you inserted another record using the default key value, and SQLite provided the next l value—10+1.

In summary, `autoincrement` prevents SQLite from recycling primary key values (`rowids`) and stops when the `rowid` reaches the maximum signed 64-bit integer value. This feature was added for specific applications that required this behavior. Unless you have such a specific need in your application, it is perhaps best to just use `integer primary key` for autoincrement columns.

Like `unique` constraints, `primary key` constraints can be defined over multiple columns. You don't have to use an integer value for your primary key. If you choose to use another value, SQLite will still maintain the `rowid` column internally, but it will also place a `unique` constraint on your declared primary key. Here's an example:

```
sqlite> create table pkey(x text, y text, primary key(x,y));
sqlite> insert into pkey values ('x','y');
sqlite> insert into pkey values ('x','x');
sqlite> select rowid, x, y from pkey;
```

```
rowid       x           y
----------  ----------  ----------
1           x           y
2           x           x

sqlite> insert into pkey values ('x','x');
SQL error: columns x, y are not unique
```

The primary key here is technically just a `unique` constraint across two columns, because SQLite will always still maintain an internal `rowid`. But many database design experts will encourage you to use real columns for your primary keys, and I encourage you to do that wherever it makes sense.

Domain Integrity

The simplest definition of domain integrity is the conformance of a column's values to its assigned domain. That is, every value in a column should exist within that column's defined domain. However, the term *domain* is a little vague. Domains are often compared to types in programming languages, such as strings or floats. And although that is not a bad analogy, domain integrity is actually much broader than that.

Domain constraints make it possible for you to start with a simple type—such as an integer—and add additional constraints to create a more restricted set of acceptable values for a column. For example, you can create a column with an integer type and add the constraint that only three such values are allowed: {-1, 0. 1}. In this case, you have modified the range of acceptable values (from the domain of all integers to just three integers), but not the data type itself. You are dealing with two things: a type and a range.

Consider another example: the `name` column in the `contacts` table. It is declared as follows:

```
name text not null collate nocase
```

The domain `text` defines the type and initial range of acceptable values. Everything following it serves to restrict and qualify that range even further. The `name` column is then the domain of all `text` values that do not include `NULL` values where uppercase letters and lowercase letters have equal value. It is still text and operates as `text`, but its range of acceptable values is further restricted from that of `text`.

You might say that a column's domain is not the same thing as its type. Rather, its domain is a combination of two things: a type and a range. The column's type defines the representation and operators of its values—how they are stored and how you can operate on them—sort, search, add, subtract, and so forth. A column's range is its set of acceptable values you can store in it, which is not necessarily the same as its declared type. The type's range represents a maximum range of values. The column's range—as you have seen—can be restricted through constraints. So for all practical purposes, you can think of a column's domain as a type with constraints tacked on.

Similarly, there are essentially two components to domain integrity: type checking and range checking. Although SQLite supports many of the standard domain constraints for range checking (`not null`, `check`, and so on), its approach to type checking is where things diverge from other databases. In fact, SQLite's approach to types and type checking is one of its most controversial, misunderstood, and disputed features. We'll cover the basics shortly and go into greater depths in Chapter 11 when we discuss SQLite internals.

For now, let's cover the easy stuff first: default values, `not null` constraints, `check` constraints, and collations.

Default Values

The `default` keyword provides a default value for a column if one is not provided in an `insert` command. The `default` keyword is only a constraint in that it prevents the absence of a value, stepping in when needed. However, it does fall within domain integrity because it provides a policy for handling `null` values in a column. If a column doesn't have a default value and you don't provide a value for it in an `insert` statement, then SQLite will insert `null` for that column. For example, `contacts.name` has a default value of `'UNKNOWN'`. With this in mind, consider the following example:

```
sqlite> insert into contacts (name) values ('Jerry');
sqlite> select * from contacts;

id          name        phone
----------  ----------  ----------
1           Jerry       UNKNOWN
```

The `insert` command inserted a row, specifying a value for `name` but not `phone`. As you can see from the resulting row, the default value for `phone` kicked in and provided the string `'UNKNOWN'`. If `phone` did not have a default value, then in this example, the value for `phone` in this row would have been `NULL` instead.

`default` also accepts three predefined ANSI/ISO reserved words for generating default dates and times. `current_time` will generate the current local time in ANSI/ISO-8601 time format (HH:MM:SS). `current_date` will generate the current date (in YYYY-MM-DD format). `current_timestamp` will produce a combination of these two (in `YYYY-MM-DD HH:MM:SS` format). Here's an example:

```
create table times ( id int,
  date not null default current_date,
  time not null default current_time,
  timestamp not null default current_timestamp );
insert into times (id) values (1);
insert into times (id) values (2);
select * from times;
```

```
id   date        time        timestamp
---  ----------  ----------  -------------------
1    2010-06-15  23:30:25    2010-06-15 23:30:25
2    2010-06-15  23:30:40    2010-06-15 23:30:40
```

These defaults come in quite handy for tables that need to log or timestamp events.

NOT NULL Constraints

If you are one of those people who is not fond of `NULL`, then the `NOT NULL` constraint is for you. `NOT NULL` ensures that values in the column may never be `NULL`. `insert` commands may not add `NULL` in the column, and `update` commands may not change existing values to `NULL`. Oftentimes, you will see `NOT NULL` raise its ugly head in `insert` statements. Specifically, a `NOT NULL` constraint without a `default` constraint will prevent any unspecified values from being used in the `insert` (because the default values provided in this case are `NULL`). In the preceding example, the `NOT` NULL constraint on `name` requires that an `insert` command always provide a value for that column. Here's an example:

```
sqlite> insert into contacts (phone) values ('555-1212');
SQL error: contacts.name may not be NULL
```

This `insert` command specified a `phone` value but not a name. The `NOT NULL` constraint on `name` kicked in and forbade the operation.

A pragmatic way to deal with unknown data and `NOT NULL` constraints is to also include a `default` constraint for the column. This is the case for `phone`. Although `phone` has a `NOT NULL` constraint, it has a `default` constraint as well. If an `insert` command does not specify a value for phone, the `default` constraint steps in and provides the value `'UNKNOWN'`, thus satisfying the `NOT NULL` constraint. To this end, people often use `default` constraints in conjunction with `NOT NULL` constraints so that `insert` commands can safely use default values while at the same time keeping `NULL` out of the column.

Check Constraints

Check constraints allow you to define expressions to test values whenever they are inserted into or updated within a column. If the values do not meet the criteria set forth in the expression, the database issues a constraint violation. Thus, it allows you to define additional data integrity checks beyond `unique` or `NOT NULL` to suit your specific application. An example of a `check` constraint might be to ensure that the value of a phone number field is at least seven characters long. To do this, you can add the constraint to the column definition of `phone` or as a stand-alone constraint in the table definition as follows:

```
create table contacts
( id integer primary key,
name text not null collate nocase,
phone text not null default 'UNKNOWN',
unique (name,phone),
check (length(phone)>=7) );
```

Here, any attempt to insert or update a value for `phone` less than seven characters will result in a constraint violation. You can use any expression in a `check` constraint that you would in a `where` clause, with the exception of subqueries. For example, say you have the table `foo` defined as follows:

```
create table foo
( x integer,
y integer check (y>x),
z integer check (z>abs(y)) );
```

In this table, every value of `z` must always be greater than `y`, which in turn must be greater than `x`. To show illustrate this, try the following:

```
insert into foo values (-2, -1, 2);
insert into foo values (-2, -1, 1);
SQL error: constraint failed

update foo set y=-3 where x=-3;
SQL error: constraint failed
```

The `check` constraints for all columns are evaluated before any modification is made. For the modification to succeed, the expressions for all constraints must evaluate to true.

Functionally, triggers can be used just as effectively as check constraints for data integrity. In fact, triggers can do much more. If you find that you can't quite express what you need in a `check` constraint, then triggers are a good alternative. Triggers are covered later in this chapter in the section "Triggers."

Foreign Key Constraints

SQLite supports the concept of relational integrity from relational theory. Like most databases, it supports this through the mechanism of foreign key constraints. Relational integrity, and by extension foreign keys, ensures that where a key value in one table logically refers to data in another table, the data in the other table *actually exists*. Classic examples are parent-child relationships, master-detail relationships for things such as orders and line items, and even episode-food relationships.

SQLite supports foreign key creation in the `create table` statement using the following syntax (simplified here for easy reading):

```
create table table_name
( column_definition references foreign_table (column_name)
  on {delete|update} integrity_action
  [not] deferrable [initially {deferred|immediate}, ]
… );
```

That syntax looks daunting, but in reality it can be broken down into three basic components. Let's use a real example of `foods` database. The `foods` and `food_types` tables are currently defined as follows:

```
CREATE TABLE food_types(
  id integer primary key,
  name text );

CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
```

We know each `food_type` has an `id` that uniquely identifies it (see the earlier discussion on primary keys). The `foods` table uses the `type_id` column to reference foods in the `food_types` table. If you want to use referential integrity to have SQLite protect your data and ensure that there always exists a food type for any reference of a food, then you can create the `foods` table like this instead:

```
create table foods(
  id integer primary key,
  type_id integer references food_types(id)
  on delete restrict
  deferrable initially deferred,
  name text );
```

The differences are shown in bold and are easy to understand if taken one at a time. The first part of the foreign key instructs SQLite that the `type_id` column references the `id` column of the `food_types` table. From there, you move on to the integrity action clause, which does most of the hard work. In this example, you've used the option `on delete restrict`. This instructs SQLite to prevent any deletion from the `food_types` table that would leave a food in the `foods` table without a parent food `id`. `restrict` is one of five possible actions you can define. The full set is as follows:

Download from Wow! eBook <www.wowebook.com>

`set null`: Change any remaining child value to `NULL` if the parent value is removed or changed to no longer exist.

`set default`: Change any remaining child value to the column default if the parent value is removed or changed to no longer exist.

`cascade`: Where the parent key is updated, update all child keys to match. Where it is deleted, delete all child rows with the key. Pay particular attention to this option, because cascading deletes can surprise you when you least expect them.

`restrict`: Where the update or delete of the parent key would result in orphaned child keys, prevent (abort) the transaction.

`no action`: Take a laid-back approach, and watch changes fly by without intervening. Only at the end of the entire statement (or transaction if the constraint is deferred) is an error raised.

Lastly, SQLite supports the `deferrable` clause, which controls whether the constraint as defined will be enforced immediately or deferred until the end of the transaction.

Collations

Collation refers to how text values are compared. Different collations employ different comparison methods. For example, one collation might be case insensitive, so the strings `'JujyFruit'` and `'JUJYFRUIT'` are considered the same. Another collation might be case sensitive, in which case the strings would be considered different.

SQLite has three built-in collations. The default is `binary`, which compares text values byte by byte using a specific C function called `memcmp()`. This happens to work nicely for many Western languages such as English. `nocase` is basically a case-insensitive collation for the 26 ASCII characters used in Latin alphabets. Finally there is `reverse`, which is the reverse of the `binary` collation. `Reverse` is more for testing (and perhaps illustration) than anything else.

The SQLite C API provides a way to create custom collations. This feature allows developers to support languages and/or locales that are not well served by the binary collation. See Chapter 7 for more information.

The `collate` keyword defines the collation for a column. For example, the collation for `contacts.name` is defined as `nocase`, which means that it is case insensitive. Thus, if you try to insert another row with a `name` value of `'JERRY'` and a `phone` value of `'555-1212'`, it should fail:

```
sqlite> insert into contacts (name,phone) values ('JERRY','555-1212');
SQL error: columns name, phone are not unique
```

According to `name`'s collation, `'JERRY'` is the same as `'Jerry'`, and there is already a row with that value. Therefore, a new row with `name='JERRY'` would be a duplicate value. By default, collation in SQLite is case sensitive. The previous example would have worked had we not defined `nocase` on `name`.

Storage Classes

As mentioned earlier, SQLite does not work like other databases when it comes to handling data types. It differs in the types it supports and in how they are stored, compared, enforced, and assigned. I'll cover the basics of SQLite storage classes next to give you a good working knowledge. In Chapter 11, I will

address many of the internals of SQLite's radically different but surprisingly flexible approach to data types.

Internally, SQLite has five primitive data types, which are referred to as *storage classes*. The term *storage class* refers to the format in which a value is stored on disk. Regardless, it is still synonymous with type, or data type. Table 4-1 describes the five storage classes.

Table 4-1. *SQLite Storage Classes*

Name	Description
`integer`	Integer values are whole numbers (positive and negative). They can vary in size: 1, 2, 3, 4, 6, or 8 bytes. The maximum integer range (8 bytes) is {-9223372036854775808,-1,0,1, 9223372036854775807}. SQLite automatically handles the integer sizes based on the numeric value.
`real`	Real values are real numbers with decimal values. SQLite uses 8-byte floats to store real numbers.
`text`	Text is character data. SQLite supports various character encodings, which include UTF-8 and UTF-16 (big and little endian). The maximum string value in SQLite is adjustable at compile time and at runtime and defaults to 1,000,000,000 bytes.
`blob`	Binary large object (`BLOB`) data is any kind of data. The maximum size for `BLOB`s in SQLite is adjustable at compile time and at runtime and defaults to 1,000,000,000 bytes.
`NULL`	`NULL` represents missing information. SQLite has full support for `NULL` handling.

SQLite infers a value's type from its representation. The following inference rules are used to do this:

- A value specified as a literal in SQL statements is assigned class `text` if it is enclosed by single or double quotes.
- A value is assigned class `integer` if the literal is specified as an unquoted number with no decimal point or exponent.
- A value is assigned class `real` if the literal is an unquoted number with a decimal point or an exponent.
- A value is assigned class `NULL` if its value is `NULL`.
- A value is assigned class `blob` if it is of the format `x'ABCD'`, where `ABCD` are hexadecimal numbers. The `x` prefix and values can be either uppercase or lowercase.

The `typeof()` SQL function returns the storage class of a value based on its representation. Using this function, the following SQL illustrates type inference in action:

```
sqlite> select typeof(3.14), typeof('3.14'),
        typeof(314), typeof(x'3142'), typeof(NULL);

typeof(3.14)  typeof('3.14')  typeof(314)  typeof(x'3142')  typeof(NULL)
------------  --------------  -----------  ---------------  ------------
real          text            integer      blob             null
```

Here are all of the five internal storage classes invoked by specific representations of data. The value 3.14 looks like a `real` and therefore is a `real`. The value `'3.14'` looks like `text` and therefore is `text`, and so on.

A single column in SQLite may contain different values *of different storage classes*. This is the first difference in SQLite data handling that usually makes those familiar with other databases sit up and say, "What?" Consider the following example:

```
sqlite> drop table domain;
sqlite> create table domain(x);
sqlite> insert into domain values (3.142);
sqlite> insert into domain values ('3.142');
sqlite> insert into domain values (3142);
sqlite> insert into domain values (x'3142');
sqlite> insert into domain values (null);
sqlite> select rowid, x, typeof(x) from domain;

rowid       x           typeof(x)
----------  ----------  ----------
1           3.142       real
2           3.142       text
3           3142        integer
4           1B          blob
5           NULL        null
```

This raises a few questions. How are the values in a column sorted or compared? How do you sort a column with `integer`, `real`, `text`, `blob`, and `NULL` values? How do you compare an `integer` with a `blob`? Which is greater? Can they ever be equal?

As it turns out, values in a column with different storages classes can be sorted. And they can be sorted because they can be compared. SQLite implements well-defined rules to do so. Storage classes are sorted by using their respective *class values*, which are defined as follows:

1. The `NULL` storage class has the lowest class value. A value with a `NULL` storage class is considered less than any other value (including another value with storage class `NULL`). Within `NULL` values, there is no specific sort order.

2. `integer` or `real` storage classes have higher value than `NULL`s and share equal class value. `integer` and `real` values are compared numerically.

3. The `text` storage class has higher value than `integer` or `real`. A value with an `integer` or a `real` storage class will always be less than a value with a `text` storage class no matter its value. When two `text` values are compared, the comparison is determined by the collation defined for the values.

4. The `blob` storage class has the highest value. Any value that is not of class `blob` will always be less than a value of class `blob`. `blob` values are compared using the C function `memcmp()`.

When SQLite sorts a column, it first groups values according to storage class—first `NULL`s, then `integer`s and `real`s, next `text`, and finally `blob`s. It then sorts the values within each group. `NULL`s are not ordered at all, `integer`s and `real`s are compared numerically, `text` is arranged by the appropriate collation, and `blob`s are sorted using `memcmp()`, effectively comparing them in a bitwise fashion. Figure 4-1 illustrates a hypothetical column sorted in ascending order.

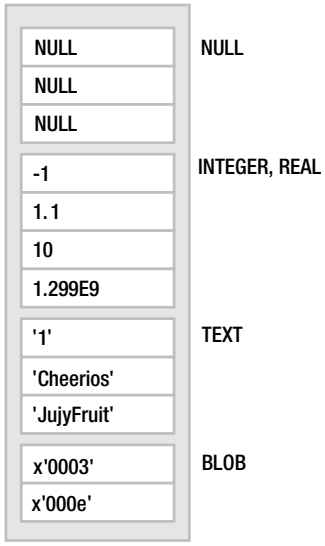


Figure 4-1. *Storage class sort order*

It's probably worth reading this section again once you've had a chance to practice some SQL using SQLite's storage classes, just to reinforce this particular aspect of SQLite. You'll return to this topic in Chapter 11, when you'll delve into the internals of storage classes, manifest typing, type affinity, and other under-the-hood topics related to types and storage classes.

Views

Views are virtual tables. They are also known as *derived tables*, because their contents are derived from the results of queries on other tables. Although views look and feel like base tables, they aren't. The contents of base tables are persistent, whereas the contents of views are dynamically generated when they are used. The syntax to create a view is as follows:

```
create view name as select-stmt;
```

The name of the view is given by ***name*** and its definition by ***select-stmt***. The resulting view will look like a base table named `name`. Imagine you had a query you ran all the time, so much that you get sick of writing it. Views are the solution for this particular chore. Say your query was as follows:

```
select f.name, ft.name, e.name
from foods f
inner join food_types ft on f.type_id=ft.id
inner join foods_episodes fe on f.id=fe.food_id
inner join episodes e on fe.episode_id=e.id;
```

This returns the name of every food, its type, and every episode it was in. It is one big table of 504 rows with just about every food fact. Rather than having to write out (or remember) the previous query every time you want these results, you can tidily restate it in the form of a view. Let's name it `details`:

```
create view details as
select f.name as fd, ft.name as tp, e.name as ep, e.season as ssn
from foods f
inner join food_types ft on f.type_id=ft.id
inner join foods_episodes fe on f.id=fe.food_id
inner join episodes e on fe.episode_id=e.id;
```

Now you can query `details` just as you would a table. Here's an example:

```
sqlite> select fd as Food, ep as Episode
        from details where ssn=7 and tp like 'Drinks';

Food                Episode
------------------  ------------------
Apple Cider         The Bottle Deposit 1
Bosco               The Secret Code
Cafe Latte          The Postponement
Cafe Latte          The Maestro
Champagne Coolies   The Wig Master
Cider               The Bottle Deposit 2
Hershey's           The Secret Code
Hot Coffee          The Maestro
Latte               The Maestro
Mellow Yellow soda  The Bottle Deposit 1
Merlot              The Rye
Orange Juice        The Wink
Tea                 The Hot Tub
Wild Turkey         The Hot Tub
```

The contents of views are dynamically generated. Thus, every time you use `details`, its associated SQL will be reexecuted, producing results based on the data in the database at that moment. Some features of views available in other databases, like view-based security, are not generally available. Some view-based security is available if you program with SQLite, using its operational control facilities. This is covered in Chapters 5 and 6.

Finally, to drop a view, use the `DROP VIEW` command:

```
drop view name;
```

The name of the view to drop is given by `name`.

UPDATABLE VIEWS

The relational model calls for updatable views. These are views that you can modify. You can run `insert` or `update` statements on them, for example, and the respective changes are applied directly to their underlying tables. Updatable views are not supported in SQLite. However, using triggers, you can create something like them. We show the technique later in the section "Triggers."

Indexes

Indexes are a construct designed to speed up queries under certain conditions. Consider the following query:

```
SELECT * FROM foods WHERE name='JujyFruit';
```

When a database searches for matching rows, the default method it uses to perform this is called a *sequential scan*. That is, it literally searches (or scans) every row in the table to see whether its `name` attribute matches `'JujyFruit'`.

However, if this query is used frequently and the `foods` table is very large, it makes far more sense to use an index approach to finding the data. SQLite uses B-tree indexes, similar to many other relational databases.

Indexes also increase the size of the database. They literally keep a copy of all columns they index. If you index every column in a table, you effectively double the size of the table. Another consideration is that indexes must be maintained. When you insert, update, or delete records, in addition to modifying the table, the database must modify each and every index on that table as well. So although indices can make queries run much faster, they can slow down inserts, updates, and similar operations.

The command to create an index is as follows:

```
create index [unique] index_name on table_name (columns)
```

The variable `index_name` is the name of the index, and `table_name` is the name of the table containing the column(s) to index. The variable `columns` is either a single column or a comma-separated list of columns.

If you use the `unique` keyword, then the index will have the added constraint that all values in the index must be unique. This applies to both the index and, by extension, to the column or columns it indexes. The `unique` constraint covers all columns defined in the index, and it is their combined values (not individual values) that must be unique. Here's an example:

```
sqlite> create table foo(a text, b text);
sqlite> create unique index foo_idx on foo(a,b);
sqlite> insert into foo values ('unique', 'value');
```

```
sqlite> insert into foo values ('unique', 'value2');
sqlite> insert into foo values ('unique', 'value');
SQL error: columns a, b are not unique
```

You can see here that uniqueness is defined by both columns collectively, not individually. Notice that collation plays an important role here as well.

To remove an index, use the `drop index` command, which is defined as follows:

```
drop index index_name;
```

Collations

Each column in the index can have a collation associated with it. For example, to create a case-insensitive index on `foods.name`, you'd use the following:

```
create index foods_name_idx on foods (name collate nocase);
```

This means that values in the `name` column will sort without respect to case. You can list the indexes for a table in the SQLite command-line program by using the `.indices` shell command. Here's example:

```
sqlite> .indices foods
foods_name_idx
For more information, you can use the .schema shell command as well:
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

You can also obtain this information by querying the `sqlite_master` table, described later in this section.

Index Utilization

It is important to understand when indexes are used and when they aren't. There are very specific conditions in which SQLite will decide to use an index. SQLite will use a single column index, if available, for the following expressions in the `WHERE` clause:

```
column {=|>|>=|<=|<} expression
expression {=|>|>=|<=|<} column
column IN (expression-list)
column IN (subquery)
```

Multicolumn indexes have more specific conditions before they are used. This is perhaps best illustrated by example. Say you have a table defined as follows:

```
create table foo (a,b,c,d);
```

Furthermore, you create a multicolumn index as follows:

```
create index foo_idx on foo (a,b,c,d);
```

The columns of `foo_idx` can be used only sequentially from left to right. That is, in the following query:

```
select * from foo where a=1 and b=2 and d=3
```

only the first and second conditions will use the index. The reason the third condition was excluded is because there was no condition that used `c` to bridge the gap to `d`. Basically, when SQLite uses a multicolumn index, it works from left to right column-wise. It starts with the left column and looks for a condition using that column. It moves to the second column, and so on. It continues until either it fails to find a valid condition in the `WHERE` clause that uses it or there are no more columns in the index to use.

But there is one more requirement. SQLite will use a multicolumn index only if all of the conditions use either the equality (`=`) or `IN` operator for all index columns *except for the rightmost index column*. For that column, you can specify up to two inequalities to define its upper and lower bounds. Consider this example:

```
select * from foo where a>1 and b=2 and c=3 and d=4
```

SQLite will only do an index scan on column `a`. The `a>1` expression becomes the rightmost index column because it uses the inequality. All columns after it are not eligible to be used as a result. Similarly, the following:

```
select * from foo where a=1 and b>2 and c=3 and d=4
```

will use the index columns `a` and `b` and stop there as `b>2` becomes the rightmost index term by its use of an inequality operator.

Lastly, when you create an index, have a reason for creating it. Make sure there is a specific performance gain you are getting before you take on the overhead that comes with it. Well-chosen indexes are a wonderful thing. Indexes that are thoughtlessly scattered here and there in the vain hope of performance are of dubious value.

Triggers

Triggers execute specific SQL commands when specific database events transpire on specific tables. The general syntax for creating a trigger is as follows:

```
create [temp|temporary] trigger name
[before|after] [insert|delete|update|update of columns] on table
action
```

A trigger is defined by a name, an action, and a table. The action, or trigger body, consists of a series of SQL commands. Triggers are said to *fire* when such events take place. Furthermore, triggers can be made to fire before or after the event using the `before` or `after` keyword, respectively. Events include `delete`, `insert`, and `update` commands issued on the specified table. Triggers can be used to create your own integrity constraints, log changes, update other tables, and many other things. They are limited only by what you can write in SQL.

Update Triggers

`Update` triggers, unlike `insert` and `delete` triggers, may be defined for specific columns in a table. The general form of this kind of trigger is as follows:

```
create trigger name
[before|after] update of column on table
action
```

The following is a SQL script that shows an `UPDATE` trigger in action:

```
create temp table log(x);

create temp trigger foods_update_log update of name on foods
begin
  insert into log values('updated foods: new name=' || new.name);
end;

begin;
update foods set name='JUJYFRUIT' where name='JujyFruit';
select * from log;
rollback;
```

This script creates a temporary table called `log`, as well as a temporary `update` trigger on `foods.name` that inserts a message into `log` when it fires. The action takes place inside the transaction that follows. The first step of the transaction updates the `name` column of the row whose `name` is `'JUJYFRUIT'`. This causes the `update` trigger to fire. When it fires, it inserts a record into the log. Next, the transaction reads the log, which shows that the trigger did indeed fire. The transaction then rolls back the change, and when the session ends, the `log` table and the `update` trigger are destroyed. Running the script produces the following output:

```
# sqlite3 foods.db < trigger.sql
```

```
create temp table log(x);

create temp trigger foods_update_log after update of name on foods
begin
  insert into log values('updated foods: new name=' || new.name);
end;

begin;
update foods set name='JUJYFRUIT' where name='JujyFruit';
SELECT * FROM LOG;
x
--------------------------------------------------
updated foods: new name=JUJYFRUIT
rollback;
```

SQLite provides access to both the old (original) row and the new (updated) row in `update` triggers. The old row is referred to as `old` and the new row as `new`. Notice in the script how the trigger refers to `new.name`. All attributes of both rows are available in `old` and `new` using the dot notation. You could have just as easily recorded `new.type_id` or `old.id`.

Error Handling

Defining a trigger before an event takes place gives you the opportunity to stop the event from happening and, equally, examining the event afterward allows you to have second thoughts. `before` and `after` triggers enable you to implement new integrity constraints. SQLite provides a special SQL function for triggers called `raise()`, which allows them to raise an error within the trigger body. `raise` is defined as follows:

```
raise(resolution, error_message);
```

The first argument is a conflict resolution policy (`abort`, `fail`, `ignore`, `rollback`, and so on). The second argument is an error message. If you use `ignore`, the remainder of the current trigger along with the SQL statement that caused the trigger to fire, as well as any subsequent triggers that would have been fired, are all terminated. If the SQL statement that caused the trigger to fire is itself part of another trigger, then that trigger resumes execution at the beginning of the next SQL command in the trigger action.

Updatable Views

Triggers make it possible to create something like updatable views, as mentioned earlier in this chapter. The idea here is that you create a view and then create a trigger that handles update events on that view. SQLite supports triggers on views using the `instead of` keywords in the trigger definition. To illustrate this, let's create a view that combines `foods` with `food_types`:

```
create view foods_view as
  select f.id fid, f.name fname, t.id tid, t.name tname
  from foods f, food_types t;
```

This view joins the two tables according to their foreign key relationship. Notice that you have created aliases for all column names in the view. This allows you to differentiate the respective `id` and `name` columns in each table when you reference them from inside the trigger. Now, let's make the view updatable by creating an `UPDATE` trigger on it:

```
create trigger on_update_foods_view
instead of update on foods_view
for each row
begin
   update foods set name=new.fname where id=new.fid;
   update food_types set name=new.tname where id=new.tid;
end;
```

Now if you try to update the `foods_view`, this trigger gets called. The trigger simply takes the values provided in the `UPDATE` statement and uses them to update the underlying base tables `foods` and `food_types`. Testing it yields the following:

Download from Wow! eBook <www.wowebook.com>

```
.echo on
-- Update the view within a transaction
begin;
update foods_view set fname='Whataburger', tname='Fast Food' where fid=413;
-- Now view the underlying rows in the base tables:
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
-- Roll it back
rollback;
-- Now look at the original record:
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
```

```
begin;
update foods_view set fname='Whataburger', tname='Fast Food' where fid=413;
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
id   type_id  name            id   name
---  -------  --------------  ---  ---------
413  1        Whataburger     1    Fast Food

rollback;

select * from foods f, food_types t where f.type_id=t.id and f.id=413;
id   type_id  name            id   name
---  -------  --------------  ---  -------
413  1        Cinnamon Bobka  1    Bakery
```

You can just as easily add `insert` and `delete` triggers to complete the trigger based manipulation of data via views.

Transactions

Transactions define boundaries around a group of SQL commands such that they either all successfully execute together or not at all. This is typically referred to as the *atomic* principle of database integrity. A classic example of the rationale behind transactions is a money transfer. Say a bank program is transferring money from one account to another. The money transfer program can do this in one of two ways: first insert (credit) the funds into account 2 and then delete (debit) it from account 1, or first delete it from account 1 and insert it into account 2. Either way, the transfer is a two-step process: an `insert` followed by a `delete`, or a `delete` followed by an `insert`.

But what happens if, during the transfer, the database server suddenly crashes or the power goes out, and the second operation does not complete? Now the money either exists in both accounts (the first scenario) or has been completely lost altogether (second scenario). Either way, someone is not going to be happy. And the database is in an inconsistent state. The point here is that these two operations must either happen together or not at all. That is the essence of transactions.

Transaction Scopes

Transactions are issued with three commands: `begin`, `commit`, and `rollback`. `begin` starts a transaction. Every operation following a `begin` can be potentially undone and will be undone if a `commit` is not issued

before the session terminates. The `commit` command commits the work performed by all operations since the start of the transaction. Similarly, the `rollback` command undoes all the work performed by all operations since the start of the transaction. A transaction is a scope in which operations are performed and committed, or rolled back, together. Here is an example:

```
sqlite> begin;
sqlite> delete from foods;
sqlite> rollback;
sqlite> select count(*) from foods;

count(*)
--------
412
```

We started a transaction, deleted all the rows in `foods`, changed our mind, and reversed those changes by issuing a `rollback`. The `select` statement shows that nothing was changed.

By default, every SQL command in SQLite is run under its own transaction. That is, if you do not define a transaction scope with `begin…commit/rollback`, SQLite will implicitly wrap every individual SQL command with a `begin…commit/rollback`. In that case, every command that completes successfully is committed. Likewise, every command that encounters an error is rolled back. This mode of operation (implicit transactions) is referred to as *autocommit mode*: SQLite automatically runs each command in its own transaction, and if the command does not fail, its changes are automatically committed.

SQLite also supports the `savepoint` and `release` commands. These commands extend the flexibility of transactions so that a body of work that incorporates multiple statements can set a `savepoint`, which SQLite can then revert to in the event of a rollback. Creating a `savepoint` is a simple as issuing the `savepoint` command with a name of your choice, just as in this next example:

```
savepoint justincase;
```

Later, if we realize our processing needs to be reverted, instead of rolling all the way back to the start of the transaction, we can use a named rollback as follows:

```
rollback [transaction] to justincase;
```

I've chosen `justincase` as the `savepoint` name. You can choose any name you like.

Conflict Resolution

As you've seen in previous examples, constraint violations cause the command that committed the violation to terminate. What exactly happens when a command terminates in the middle of making a bunch of changes to the database? In most databases, all of the changes are undone. That is the way the database is programmed to handle a constraint violation—end of story.

SQLite, however, has an uncommon feature that allows you to specify different ways to handle (or recover from) constraint violations. It is called *conflict resolution*. Take, for example, the following `UPDATE`:

```
sqlite> update foods set id=800-id;
SQL error: PRIMARY KEY must be unique
```

This results in a `UNIQUE` constraint violation because once the `update` statement reaches the 388th record, it attempts to update its `id` value to 800-388=412. But a row with an `id` of 412 already exists, so it aborts the command. But SQLite already updated the first 387 rows before it reached this constraint violation.

What happens to them? The default behavior is to terminate the command and reverse all the changes it made, while leaving the transaction intact.

But what if you wanted these 387 changes to stick despite the constraint violation? Well, believe it or not, you can have it that way too, if you want. You just need to use the appropriate conflict resolution. There are five possible resolutions, or policies, that can be applied to address a conflict (constraint violation): `replace`, `ignore`, `fail`, `abort`, and `rollback`. These five resolutions define a spectrum of error tolerance or sensitivity: from `replace`, the most forgiving, to `rollback`, the most strict. The resolutions are defined as follows in order of their severity:

1. `replace`: When a `unique` constraint violation is encountered, SQLite removes the row (or rows) that caused the violation and replaces it (them) with the new row from the `insert` or `update`. The SQL operation continues without error. If a `NOT NULL` constraint violation occurs, the `NULL` value is replaced by the default value for that column. If the column has no default value, then SQLite applies the `abort` policy. It is important to note that when this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows. This behavior, however, is subject to change in a future release.

2. `ignore`: When a constraint violation is encountered, SQLite allows the command to continue and leaves the row that triggered the violation unchanged. Other rows before and after the row in question continue to be modified by the command. Thus, all rows in the operation that trigger constraint violations are simply left unchanged, and the command proceeds without error.

3. `fail`: When a constraint violation is encountered, SQLite terminates the command but does not restore the changes it made prior to encountering the violation. That is, all changes within the SQL command up to the violation are preserved. For example, if an `update` statement encountered a constraint violation on the 100th row it attempts to update, then the changes to the first 99 rows already modified remain intact, but changes to rows 100 and beyond never occur as the command is terminated.

4. `abort`: When a constraint violation is encountered, SQLite restores all changes the command made and terminates it. `abort` is the default resolution for all operations in SQLite. It is also the behavior defined in the SQL standard. As a side note, `abort` is also the most expensive conflict resolution policy—requiring extra work even if no conflicts ever occur.

5. `rollback`: When a constraint violation is encountered, SQLite performs a `rollback`—aborting the current command along with the entire transaction. The net result is that all changes made by the current command *and all previous commands* in the transaction are rolled back. This is the most drastic level of conflict resolution where a single violation results in a complete reversal of everything performed in a transaction.

Conflict resolution can be specified within SQL commands as well as within table and index definitions. Specifically, conflict resolution can be specified in `insert`, `update`, `create table`, and `create index`. Furthermore, it has specific implications within triggers. The syntax for conflict resolution in `insert` and `update` is as follows:

```
insert or resolution into table (column_list) values (value_list);
update or resolution table set (value_list) where predicate;
```

The conflict resolution policy comes right after the `insert` or `update` command and is prefixed with `OR`. Also, the `insert or replace` expression can be abbreviated as just `replace`. This is similar to other database's "merge" or "upsert" behavior.

In the preceding `update` example, the updates made to the 387 records were rolled back because the default resolution is `abort`. If you wanted the updates to stick, you could use the `fail` resolution. To illustrate this, in the following example you copy `foods` into a new table `test` and use it as the guinea pig. You add an additional column to `test` called `modified`, the default value of which is `'no'`. In the `update`, you change this to `'yes'` to track which records are updated before the constraint violation occurs. Using the `fail` resolution, these updates will remain unchanged, and you can track afterward how many records were updated.

```
create table test as select * from foods;
create unique index test_idx on test(id);
alter table test add column modified text not null default 'no';
select count(*) from test where modified='no';

count(*)
--------------------
412

update or fail test set id=800-id, modified='yes';
SQL error: column id is not unique

select count(*) from test where modified='yes';

count(*)
--------------------
387

drop table test;
```

■ **Caution** There is one consideration with `fail` of which you need to be aware. The order that records are updated is nondeterministic. That is, you cannot be certain of the order of the records in the table or the order in which SQLite processes them. You might assume that it follows the order of the `rowid` column, but this is not a safe assumption to make. There is nothing in the documentation that says so. Once again, never assume any implicit ordering when working with any kind of database. If you are going to use `fail`, in many cases it might be better to use `ignore`. `ignore` will finish the job and modify all records that can be modified rather than bailing out on the first violation.

When defined within tables, conflict resolution is specified for individual columns. Here's an example:

```
sqlite> create temp table cast(name text unique on conflict rollback);
sqlite> insert into cast values ('Jerry');
sqlite> insert into cast values ('Elaine');
sqlite> insert into cast values ('Kramer');
```

The `cast` table has a single column `name` with a `unique` constraint and conflict resolution set to `rollback`. Any `insert` or `update` that triggers a constraint violation on `name` will be arbitrated by the `rollback` resolution rather than the default `abort`. The result will abort not only the statement but the entire transaction as well:

```
sqlite> begin;
sqlite> insert into cast values('Jerry');
SQL error: uniqueness constraint failed

sqlite> commit;
SQL error: cannot commit - no transaction is active
```

`commit` failed here because the `name`'s conflict resolution already aborted the transaction. `create index` works the same way. Conflict resolution within tables and indices changes the default behavior of the operation from `abort` to that defined for the specific columns when those columns are the source of the constraint violation.

Conflict resolution at the statement level (DML) overrides that defined at the object level (DDL). Working from the previous example:

```
sqlite> begin;
sqlite> insert or replace into cast values('Jerry');
sqlite> commit;
```

the `replace` resolution in the `insert` overrides the `rollback` resolution defined on `cast.name`.

Database Locks

Locking is closely associated with transactions in SQLite. To use transactions effectively, you need to know a little something about how it does locking.

SQLite has coarse-grained locking. When a session is writing to the database, all other sessions are locked out until the writing session completes its transaction. To help with this, SQLite has a locking scheme that helps defer writer locks until the last possible moment in order to maximize concurrency.

WRITE-AHEAD LOGGING: THE FUTURE OF SQLITE

Coming soon in version 3.7.0 of SQLite is write-ahead logging (WAL). This will change the behavior of transactions and locking to free SQLite of the "one writer" behavior described here. Chapter 11 covers these forthcoming changes to the internals of SQLite.

SQLite uses a lock escalation policy whereby a connection gradually obtains exclusive access to a database in order to write to it. There are five different locking *states* in SQLite: *unlocked*, *shared*, *reserved*, *pending*, and *exclusive*. Each database session (or connection) can be in only one of these states at any given time. Furthermore, there is a corresponding lock for each state, except for unlocked—there is no lock required to be in the unlocked state.

To begin with, the most basic state is unlocked. In this state, no session is accessing data from the database. When you connect to a database or even initiate a transaction with `BEGIN`, your connection is in the unlocked state.

The next state beyond unlocked is shared. For a session to read from the database (not write), it must first enter the shared state and must therefore acquire a *shared lock*. Multiple sessions can simultaneously acquire and hold shared locks at any given time. Therefore, multiple sessions can read from a common database at any given time. However, no session can write to the database during this time—while any shared locks are active.

If a session wants to write to the database, it must first acquire a *reserved lock*. Only one reserved lock may be held at one time for a given database. Shared locks can coexist with a reserved lock. A reserved lock is the first phase of writing to a database. It does not block sessions with shared locks from reading, and it does not prevent sessions from acquiring new shared locks.

Once a session has a reserved lock, it can begin the process of making modifications; *however,* these modifications are cached and not actually written to disk. The reader's changes are stored in a memory cache (see the discussion of the cache_size pragma in the section "Database Configuration," later in this chapter, for more information).

When the session wants to commit the changes (or transaction) to the database, it begins the process of promoting its reserved lock to an *exclusive lock*. To get an exclusive lock, it must first promote its reserved lock to a *pending lock*. A pending lock starts a process of attrition whereby no new shared locks can be obtained. That is, other sessions with existing shared locks are allowed to continue as normal, but other sessions cannot acquire new shared locks. At this point, the session with the pending lock is waiting for the other sessions with shared locks to finish what they are doing and release them.

Once all shared locks are released, the session with the pending lock can promote it to an exclusive lock. It is then free to make changes to the database. All of the previously cached changes are written to the database file.

Deadlocks

Although you may find the preceding discussion on locking to be interesting, you are probably wondering at this point why any of it matters. Why do you need to know about locking? Well, if you don't know what you are doing, you can end up in a deadlock.

Consider the following scenario illustrated in Table 4-2. Two sessions, A and B—completely oblivious to one another—are working on the same database at the same time. Session A issues the first command, B the second and third, A the fourth, and so on.

Table 4-2. *A Portrait of a Deadlock*

Session A	Session B
`sqlite> begin;`	
	`sqlite> begin;`
	`sqlite> insert into foo values ('x');`
`sqlite> select * from foo;`	
	`sqlite> commit;`
	`SQL error: database is locked`
`sqlite> insert into foo values ('x');`	
`SQL error: database is locked`	

Both sessions wind up in a deadlock. Session B was the first to try to write to the database and therefore has a pending lock. A attempts to write but fails when `INSERT` tries to promote its shared lock to a reserved lock.

For the sake of argument, let's say that A decides to just wait around for the database to become writable. So does B. Then at this point, everyone else is effectively locked out too. If you try to open a third session, it won't even be able to read from the database. The reason is that B has a pending lock, which prevents any sessions from acquiring shared locks. So, not only are A and B deadlocked, they have locked everyone else out of the database as well. Basically, you have a shared lock and one pending lock that don't want to relinquish control, and until one does, nobody can do anything.

How do you avoid a deadlock? It's not like A and B can sit down in a meeting and work it out with their lawyers. A and B don't even know each other exists. The answer is to *pick the right transaction type for the job.*

Transaction Types

SQLite has three different transaction types that start transactions in different locking states. Transactions can be started as `deferred`, `immediate`, or `exclusive`. A transaction's type is specified in the `begin` command:

```
begin [ deferred | immediate | exclusive ] transaction;
```

A *deferred transaction* does not acquire any locks until it has to. Thus, with a deferred transaction, the `begin` statement itself does nothing—it starts in the unlocked state. This is the default. If you simply issue a `begin`, then your transaction is `deferred` and therefore sitting in the unlocked state. Multiple sessions can simultaneously start `deferred` transactions at the same time without creating any locks. In this case, the first read operation against a database acquires a shared lock, and similarly the first write operation *attempts* to acquire a reserved lock.

An immediate transaction attempts to obtain a reserved lock as soon as the `begin` command is executed. If successful, `begin immediate` guarantees that no other session will be able to write to the database. As you know, other sessions can continue to read from the database, but the reserved lock prevents any new sessions from reading. Another consequence of the reserved lock is that no other sessions will be able to successfully issue a `begin immediate` or `begin exclusive` command. SQLite will return a `SQLITE_BUSY` error. During this time, you can make some modifications to the database, but you may not necessarily be able to commit them. When you call `commit`, you could get `SQLITE_BUSY`. This means that there are other readers active, as in the earlier example. Once they are gone, you can commit the transaction.

An *exclusive transaction* obtains an exclusive lock on the database. This works similarly to `immediate`, but when you successfully issue it, `exclusive` guarantees that no other session is active in the database and that you can read or write with impunity.

The crux of the problem in the preceding example is that both sessions ultimately wanted to write to the database, but they made no attempt to relinquish their locks. Ultimately, it was the shared lock that caused the problem. If both sessions had started with `begin immediate`, then the deadlock would not have occurred. In this case, only one of the sessions would have been able to enter `begin immediate` at one time, while the other would have to wait. The one that has to wait could keep retrying with the assurance that it would eventually get in. `begin immediate` and `begin exclusive`, if used by all sessions that want to write to the database, provide a synchronization mechanism, thereby preventing deadlocks. For this approach to work, though, everyone has to follow the rules.

The bottom line is this: if you are using a database that no other connections are using, then a simple `begin` will suffice. If, however, you are using a database that other connections are also writing to, both you, and they should use `begin immediate` or `begin exclusive` to initiate transactions. It works out best that way for both of you. Transactions and locks are covered in more detail in Chapter 5.

Database Administration

Database administration is generally concerned with controlling how a database operates. Many of the database administration tasks you'll want to perform are done via SQL commands. SQLite includes some unique administrative features of its own, such the means to "attach" multiple databases to a single session, as well as database *pragmas,* which can be used for setting various configuration parameters.

Attaching Databases

SQLite allows you to "attach" multiple databases to the current session using the `attach` command. When you attach a database, all of its contents are accessible in the global scope of the current database file. `attach` has the following syntax:

```
attach [database] filename as database_name;
```

Here, `filename` refers to the path and name of the SQLite database file, and `database_name` refers to the logical name with which to reference that database and its objects. The main database is automatically assigned the name `main`. If you create any temporary objects, then SQLite will create an attached database name `temp`. (You can see these objects using the `database_list` pragma, described later.) The logical name may be used to reference objects within the attached database. If there are tables or other database objects that share the same name in both databases, then the logical name is required to reference such objects in the attached database. For example, if both databases have a table called `foo`,

and the logical name of the attached database is `db2`, then the only way to query `foo` in `db2` is by using the fully qualified name `db2.foo`, as follows:

```
sqlite> attach database '/tmp/db' as db2;
sqlite> select * from db2.foo;

x
----------
bar
```

If you really want to, you can qualify objects in the main database using the name `main`:

```
sqlite> select * from main.foods limit 2;

id          type_id     name
----------  ----------  --------------
1           1           Bagels
2           1           Bagels, raisin
```

The same is true with the temporary database:

```
sqlite> create temp table foo as select * from food_types limit 3;
sqlite> select * from temp.foo;

id   name
---  -------------
1    Bakery
2    Cereal
3    Chicken/Fowl
```

You detach databases with the `detach database` command, defined as follows:

```
detach [database] database_name;
```

This command takes the logical name of the attached database (given by ***`database_name`***) and detaches the associated database file. You get a list of attached databases using the database_list pragma, explained in the section "Database Configuration."

Cleaning Databases

SQLite has two commands designed for cleaning—`reindex` and `vacuum`. `reindex` is used to rebuild indexes. It has two forms:

```
reindex collation_name;
reindex table_name|index_name;
```

The first form rebuilds all indexes that use the collation name given by `collation_name`. It is needed only when you change the behavior of a user-defined collating sequence (for example, multiple sort orders in Chinese). All indexes in a table (or a particular index given its name) can be rebuilt with the second form.

`Vacuum` cleans out any unused space in the database by rebuilding the database file. `Vacuum` will not work if there are any open transactions. An alternative to manually running `VACUUM` statements is autovacuum. This feature is enabled using the `auto_vacuum` pragma, described in the next section.

Database Configuration

SQLite doesn't have a configuration file. Rather, all of its configuration parameters are implemented using *pragmas*. Pragmas work in different ways. Some are like variables; others are like commands. They cover many aspects of the database, such as runtime information, database schema, versioning, file format, memory use, and debugging. Some pragmas are read and set like variables, while others require arguments and are called like functions. Many pragmas have both temporary and permanent forms. Temporary forms affect only the current session for the duration of its lifetime. The permanent forms are stored in the database and affect every session. The cache size is one such example.

This section covers the most commonly used pragmas. You can find a complete list of all SQLite pragmas.

The Connection Cache Size

The cache size pragmas influence how many database pages a session can hold in memory. To set the default cache size for the current session, you use the `cache_size` pragma:

```
sqlite> pragma cache_size;

cache_size
---------------
2000

sqlite> pragma cache_size=10000;
sqlite> pragma cache_size;

cache_size
---------------
10000
```

You can permanently set the cache size for all sessions using the `default_cache_size` pragma. This setting is stored in the database. This will take effect only for sessions created after the change, not for currently active sessions.

One of the uses for the cache is in storing pending changes when a session is in a `reserved` state (it has a `reserved` lock), as described earlier in the section "Transactions." If the session fills up the cache, it will not be able to continue further modifications until it gets an `exclusive` lock, which means that it may have to first wait for readers to clear.

If you or your programs perform many updates or deletes on a database that is being used by many other sessions, it may help you to increase the cache size. The larger the cache size, the more modifications a session can cache change before it has to get an `exclusive` lock. This not only allows a session to get more work done before having to wait, it also cuts down on the time the exclusive locks needs to be held, because all the work is done up front. In this case, the `exclusive` lock only needs to be held long enough to flush the changes in the cache to disk. Some tips for tuning the cache size are covered in Chapter 5.

Getting Database Information

You can obtain database information using the database schema pragmas, defined as follows:

Download from Wow! eBook <www.wowebook.com>

- `database_list`: Lists information about all attached databases.
- `index_info`: Lists information about the columns within an index. It takes an index name as an argument.
- `index_list`: Lists information about the indexes in a table. It takes a table name as an argument.
- `table_info`: Lists information about all columns in a table.

The following illustrates some information provided by these pragmas:

```
sqlite> pragma database_list;

seq   name     file
----  -------  --------------------
0     main     /tmp/foods.db
2     db2      /tmp/db

sqlite> create index foods_name_type_idx on foods(name,type_id);
sqlite> pragma index_info(foods_name_type_idx);

seqn  cid      name
----  -------  --------------------
0     2        name
1     1        type_id

sqlite> pragma index_list(foods);

seq    name                unique
-----  ------------------- ---------------
0      foods_name_type_idx 0

sqlite> pragma table_info(foods);

cid    name             type             notn  dflt  pk
-----  ---------------  ---------------  ----  ----  ----------
0      id               integer          0           1
1      type_id          integer          0           0
2      name             text             0           0
```

Synchronous Writes

Normally, SQLite commits all changes to disk at critical moments to ensure transaction durability. This is similar to the checkpoint functionality in other databases. However, it is possible to turn this off for performance gains. You do this with the `synchronous` pragma. There are three settings: `full`, `normal`, and `off`. They are defined as follows:

- `Full`: SQLite will pause at critical moments to make sure that data has actually been written to the disk surface before continuing. This ensures that if the operating system crashes or if there is a power failure, the database will be uncorrupted after rebooting. `Full` synchronous is very safe, but it is also slow.

- `Normal`: SQLite will still pause at the most critical moments but less often than in `full` mode. There is a very small (though nonzero) chance that a power failure at just the wrong time could corrupt the database in `normal` mode. But in practice, you are more likely to suffer a catastrophic disk failure or some other unrecoverable hardware fault.
- `Off`: SQLite continues operation without pausing as soon as it has handed data off to the operating system. This can speed up some operations as much as 50 or more times. If the application running SQLite crashes, the data will be safe. However, if the operating system crashes or the computer loses power, the database may be corrupted.

There is no persistent form of the `synchronous` pragma. Chapter 5 explains this setting's crucial role in transaction durability and how it works.

Temporary Storage

Temporary storage is where SQLite keeps transient data such as temporary tables, indexes, and other objects. By default, SQLite uses a compiled-in location, which varies between platforms. There are two pragmas that govern temporary storage: `temp_store` and `temp_store_directory`. The first pragma determines whether SQLite uses memory or disk for temporary storage. There are actually three possible values: `default`, `file`, or `memory`. `default` uses the compiled-in default, `file` uses an operating system file, and `memory` uses `RAM`. If `file` is set as the storage medium, then the second pragma, `temp_store_directory`, can be used to set the directory in which the temporary storage file is placed.

Page Size, Encoding, and Autovacuum

The database page size, encoding, and autovacuuming must be set before a database is created. That is, to alter the defaults, you must first set these pragmas before creating any database objects in a new database. The defaults are a page size derived from a number of host-specific factors such as disk sector size and UTF-8 encoding. SQLite supports page sizes ranging from 512 to 32,786 bytes, in powers of 2. Supported encodings are UTF-8, UTF-16le (little-endian UTF-16 encoding), and UTF-16be (big-endian UTF-16 encoding).

A database's size can be automatically kept to a minimum using the `auto_vacuum` pragma. Normally, when a transaction that deletes data from a database is committed, the database file remains the same size. When the `auto_vacuum` pragma is enabled, the database file shrinks when a transaction that deletes data is committed. To support this functionality, the database stores extra information internally, resulting in slightly larger database files than would otherwise be possible. The `vacuum` command has no effect on databases that use `auto_vacuum`.

Debugging

There are four pragmas for various debugging purposes. The `integrity_check` pragma looks for out-of-order records, missing pages, malformed records, and corrupted indexes. If any problems are found, then a single string is returned describing the problems. If everything is in order, SQLite returns `ok`. The other pragmas are used for tracing the parser and virtual database engine and can be enabled only if SQLite is compiled with debugging information. You can find detailed information on these pragmas in Chapter 11.

The System Catalog

The `sqlite_master` table is a system table that contains information about all the tables, views, indexes, and triggers in the database. For example, the current contents of the `foods` database are as follows:

```
sqlite> select type, name, rootpage from sqlite_master;

type        name                        rootpage
----------  --------------------------  ----------
table       episodes                    2
table       foods                       3
table       foods_episodes              4
table       food_types                  5
index       foods_name_idx              30
table       sqlite_sequence             50
trigger     foods_update_trg            0
trigger     foods_insert_trg            0
trigger     foods_delete_trg            0
```

The `type` column refers to the type of object, `name` is of course the name of the object, and `rootpage` refers to the first B-tree page of the object in the database file. This latter column is relevant only for tables and indexes.

The `sqlite_master` table also contains another column called `sql`, which stores the DML used to create the object. Here's an example:

```
sqlite> select sql from sqlite_master where name='foods_update_trg';

create trigger foods_update_trg
before update of type_id on foods
begin
  select case
     when (select id from food_types where id=new.type_id) is null
     then raise( abort,
                 'Foreign Key Violation: foods.type_id is not in food_types.id')
  end;
end
```

Viewing Query Plans

You can view the way SQLite goes about executing a query by using the `explain query plan` command. The `explain query plan` command lists the steps SQLite carries out to access and process tables and data to satisfy your query.

To use `explain query plan`, just issue the command followed by your normal query text. For instance, here's what `explain query plan` has to say about how a query on the `foods` table is processed:

```
sqlite> explain query plan select * from foods where id = 145;
order       from        detail
----------  ----------  ----------------------------
0           0           TABLE foods USING PRIMARY KEY
```

This means that SQLite is accessing the `foods` table and using the primary key (on `id`) to perform the access (rather than scanning the table's data in a brute-force fashion).
Studying these query plans is the key to understanding how SQLite is approaching your data and satisfying your query. You can spot when and how indices are used and the order in which tables are used in joins. This is of immense help when troubleshooting long-running queries and other issues.

Summary

SQL may be a simple language to use, but there is quite a bit of it, and it's taken us two chapters just to introduce the major concepts for SQLite's implementation of SQL. But that shouldn't be too surprising, because it is the sole interface through which to interact with a relational database. Whether you are a casual user, system administrator, or developer, you have to know SQL if you are going to work with a relational database.

If you are programming with SQLite, then you should be off to a good start on the SQL side of things. Now you need to know a little about how SQLite goes about executing all of these commands. This is where Chapter 5 should prove useful. It will introduce you to the API and show you how it works in relation to the way SQLite functions internally.

CHAPTER 5

SQLite Design and Concepts

This chapter sets the stage for the following chapters, each of which focuses on a different aspect of programming with SQLite. It addresses the things that you as a programmer should know about SQLite when using it in your code. Whether you are programming with SQLite in its native C or in your favorite scripting language, it helps to understand not only its API but also a little about its architecture and implementation. Armed with this knowledge, you will be better equipped to write code that runs faster and avoids potential pitfalls such as deadlocks or unexpected errors. You will see how SQLite works in relation to your code, and you can be more confident that you are attacking the problem from the right direction.

You don't have to comb through the bowels of the source code to understand these things, and you don't have to be a C programmer. SQLite's design and concepts are all very straightforward and easy to understand. And there are only a few things you need to know. This chapter lays out the main concepts and components on which to build your understanding.

Key to understanding SQLite is knowing how the API works. So, this chapter starts with a conceptual introduction to the API, illustrating its major data structures, its general design, and its major functions. It also looks at some of the major SQLite subsystems that play important roles in query processing.

Beyond just knowing what functions do what, we'll also look above the API, seeing how everything operates in terms of transactions. Everything involving a SQLite database is done within the context of a transaction. Then you need to look beneath the API, seeing how transactions work in terms of locks. Locks can cause problems if you don't know how they operate. By understanding locks, not only can you avoid potential concurrency problems, but you can also optimize your queries by controlling how your program uses them.

Finally, you have to understand how all of these things apply to writing code. The last part of the chapter brings all three topics—the API, transactions, and locks—together and looks at different examples of good and bad code. It specifically identifies scenarios that could cause problems and provides some insight on how to address them.

With these things in order, you will be well on your way to conquering the C API and the API of any language extension.

The API

Functionally, the SQLite API can be separated into two general parts: the core API and the extension API. The core API consists all the functions used to perform basic database operations: connecting to the database, processing SQL, and iterating through results. It also includes various utility functions that help with tasks such as string formatting, operational control, debugging, and error handling. The

extension API offers different ways to extend SQLite by creating your own user-defined SQL extensions, which you can integrate into SQLite's SQL dialect.

The Principal Data Structures

As you saw in Chapter 1, there are many components in SQLite—parser, tokenizer, virtual machine, and so on. But from a programmer's point of view, the main things to know about are connections, statements, the B-tree, and the pager. Figure 5-1 shows their relationships to one another. These objects collectively address the three principal things you must know about SQLite to write good code: the API, transactions, and locks.

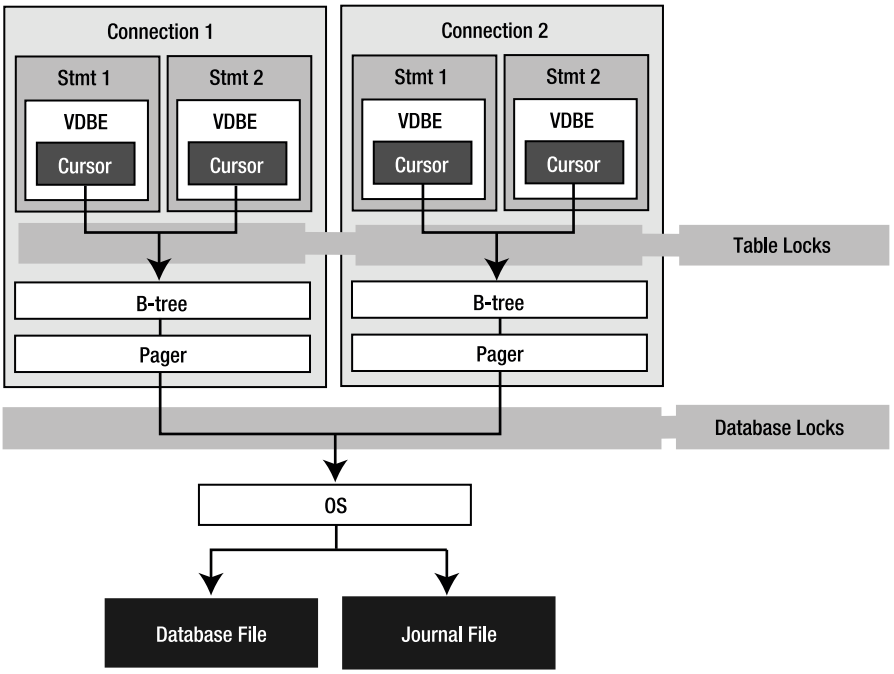


Figure 5-1. *SQLite C API object model*

Technically, the B-tree and pager are not part of the API; they are off-limits. But they play a critical role in transactions and locks. We will explore their involvement in these matters here in this section and later in the section "Transactions."

Connections and Statements

The two fundamental data structures in the API associated with query processing are the connection and the statement. In most language extensions, you will see both a connection object and a statement object, which are used to execute queries. In the C API, they correspond directly to the `sqlite3` and

`sqlite3_stmt` handles, respectively. Every major operation in the API is done using one of these two structures.

A connection represents a single connection to a database as well as a single transaction context. Statements are derived from connections. That is, every statement has an associated connection object. A statement represents a single compiled SQL statement. Internally, it is expressed in the form of VDBE byte code—a program that when executed will carry out the SQL command. Statements contain everything needed to execute a command. They include resources to hold the state of the VDBE program as it is executed in a stepwise fashion, B-tree cursors that point to records on disk, and other things such as bound parameters, which are addressed later in the section "Parameter Binding." Although they contain many different things, you can simply think of them as cursors with which to iterate through a result set, or as opaque handles referencing a single SQL command.

The B-tree and Pager

Each connection can have multiple database objects—one main database followed by any attached databases. Each database object has one B-tree object, which in turn has one pager object.

Statements use their connection's B-tree and pager objects to read and write data to and from the database. Statements that read the database iterate over B-trees using cursors. Cursors iterate over records, and records are stored in pages. As a cursor traverses records, it also traverses pages. For a cursor to access a page, it must first be loaded from disk into memory. This is the pager's job. Whenever the B-tree needs a particular page in the database, it asks the pager to fetch it from disk. The pager then loads the page into its page cache, which is a memory buffer. Once it is in the page cache, the B-tree and its associated cursor can get to the records inside the page.

If the cursor modifies the page, then the pager must also take special measures to preserve the original page in the event of a transaction rollback. Thus, the pager is responsible for reading and writing to and from the database, maintaining a memory cache or pages, and managing transactions. In addition to this, it manages locks and crash recovery. All of these responsibilities are covered later in "Transactions."

There are two things you should know about connections and transactions in general. First, when it comes to any operation on the database, a connection *always* operates under a transaction. Second, a connection *never* has more than one transaction open at a time. That means all statements derived from a given connection operate within the same transaction context. If you want the two statements to run in separate transactions, then you have to use multiple connections—one connection for each transaction context.

The Core API

As mentioned earlier, the core API is concerned with executing SQL commands. It is made of various functions for performing queries as well as various utility functions for managing other aspects of the database. There are two essential methods for executing SQL commands: prepared queries and wrapped queries. Prepared queries are the way in which SQLite ultimately executes all commands, both in the API and internally. It is a three-phase process consisting of preparation, execution, and finalization. There is a single API function associated with each phase. Associated with the execution phase are functions with which to obtain record and column information from result sets.

In addition to the standard query method, there are two wrapper functions, which wrap the three phases into a single function call. They provide a convenient way to execute a SQL command all at once. These functions are just a few of the many miscellaneous utility functions in the API. We will look at all of the query methods along with their associated utility functions in this section. Before we do however, let's first look at how to connect to a database.

Connecting to a Database

Connecting to a database involves little more than opening a file. Every SQLite database is stored in a single operating system file—one database to one file. The function used to connect, or open, a database in the C API is `sqlite3_open()` and is basically just a system call for opening a file. SQLite can also create in-memory databases. In most extensions, if you use `:memory:` or an empty string as the name for the database, it will create the database in RAM. The database will be accessible only to the connection that created it (it cannot be shared with other connections). Furthermore, the database will only last for the duration of the connection. It is deleted from memory when the connection closes.

When you connect to a database on disk, SQLite opens a file, if it exists. If you try to open a file that doesn't exist, SQLite will assume that you want to create a new database. In this case, SQLite doesn't immediately create a new operating system file. It will create a new file only if you put something into the new database—create a table or view or other database object. If you just open a new database, do nothing, and close it, SQLite does not bother with creating a database file—it would just be an empty file anyway.

There is an important reason for not creating a new file right away. Certain database options, such as encoding, page size, and autovacuum, can be set only before you create a database. By default, SQLite uses a 1,024-byte page size. However, you can use different page sizes ranging from 512 to 32,768 bytes by powers of 2. You might want to use different page sizes for performance reasons. For example, setting the page size to match the operating system's page size can make I/O more efficient. Larger page sizes can help with applications that deal with a lot of binary data. You set the database page size using the `page_size` pragma.

Encoding is another permanent database setting. You specify a database's encoding using the encoding pragma, which can be UTF-8, UTF-16, UTF-16le (little endian), and UTF-16be (big endian).

Finally there is autovacuum, which you set with the `auto_vacuum` pragma. When a transaction deletes data from a database, SQLite's default behavior is to keep the deleted pages around for recycling. The database file remains the same size. To free the pages, you must explicitly issue a `vacuum` command to reclaim the unused space. The autovacuum feature causes SQLite to automatically shrink the database file when data is deleted. This feature is often more useful in embedded applications where storage is a premium.

Once you open a database—file or memory—it will be represented internally by an opaque `sqlite3` connection handle. This handle represents a single connection to a database. Connection objects in extensions abstract this handle and sometimes implement methods that correspond to API functions that take the handle as an argument.

Executing Prepared Queries

As stated earlier, the prepared query method is the actual process by which SQLite executes all SQL commands. Executing a SQL command is a three-step process:

- **Preparation:** The parser, tokenizer, and code generator prepare the SQL statement by compiling it into VDBE byte code. In the C API, this is performed by the `sqlite3_prepare_v2()` function, which talks directly to the compiler. The compiler creates a `sqlite3_stmt` handle (statement handle) that contains the byte code and all other resources needed to execute the command and iterate over the result set (if the command produces one).
- **Execution:** The VDBE executes the byte code. Execution is a stepwise process. In the C API, each step is initiated by `sqlite3_step()`, which causes the VDBE to step through the byte code. The first call to `sqlite3_step()` usually acquires a lock of some kind, which varies according to what the command does (reads or writes). For `SELECT` statements, each call to `sqlite3_step()` positions the statement handle's cursor on the next row of the result set. For each row in the set, it returns `SQLITE_ROW` until it reaches the end, whereupon it returns `SQLITE_DONE`. For other SQL statements (`insert`, `update`, `delete`, and so on), the first call to `sqlite3_step()` causes the VDBE to process the entire command.
- **Finalization:** The VDBE closes the statement and deallocates resources. In the C API, this is performed by `sqlite3_finalize()`, which causes the VDBE to terminate the program, freeing resources and closing the statement handle.

Each step—preparation, execution, finalization—corresponds to a respective statement handle state—prepared, active, or finalized. Prepared means that all necessary resources have been allocated and the statement is ready to be executed, but nothing has been started. No lock has been acquired, nor will a lock be acquired until the first call to `sqlite3_step()`. The active state starts with the first call to `sqlite3_step()`. At that point, the statement is in the process of being executed, and some kind of lock is in play. Finalized means that the statement is closed and all associated resources have been freed. Figure 5-2 shows these steps and states.

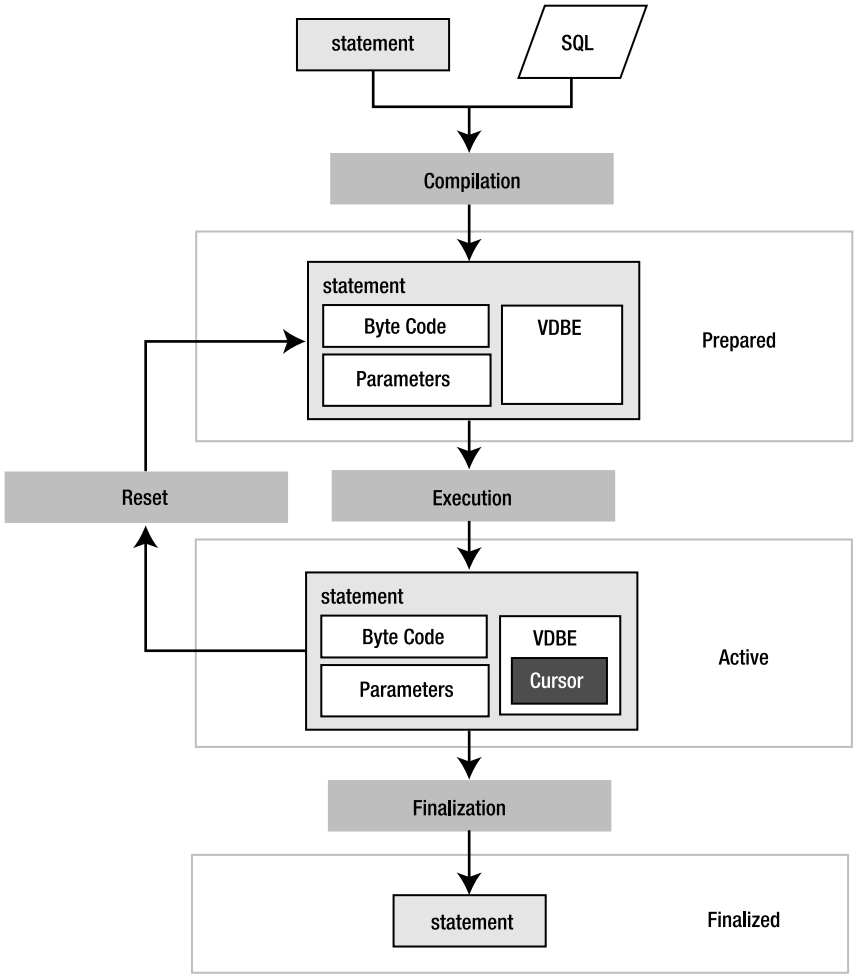


Figure 5-2. *Statement processing*

The following pseudocode illustrates the general process of executing a query in SQLite:

```
# 1. Open the database, create a connection object (db)
db = open('foods.db')

# 2.A. Prepare a statement
stmt = db.prepare('select * from episodes')
```

Download from Wow! eBook <www.wowebook.com>

```
# 2.B. Execute. Call step() is until cursor reaches end of result set.
while stmt.step() == SQLITE_ROW
  print stmt.column('name')
end

# 2.C. Finalize. Release read lock.
stmt.finalize()

# 3. Insert a record
stmt = db.prepare('INSERT INTO foods VALUES (…)')
stmt.step()
stmt.finalize()

# 4. Close database connection.
db.close()
```

This pseudocode is an object-oriented analog of the API, similar to what you might find in a scripting language. The methods all correspond to SQLite API functions. For example, `prepare()` mirrors `sqlite3_prepare()_v2`, and so on. This example performs a `SELECT`, iterating over all returned rows, followed by an `INSERT`, which is processed by a single call to `step()`.

TEMPORARY STORAGE

Temporary storage is an important part of query processing. SQLite occasionally needs to store intermediate results produced in the process of executing commands—for instance, when results need to be sorted for an `order by` clause or rows in one table are joined with rows in another table. This information is often stored in temporary storage. Temporary storage is kept either in RAM or in a file. Although SQLite has suitable defaults for all platforms, you may want to control how and where it uses this storage. The `temp_store` pragma lets you specify whether to use RAM or file-based storage. If you use file-based storage, you can use the `temp_store_directory` pragma to specify where the storage file is created.

Using Parameterized SQL

SQL statements can contain parameters. Parameters are placeholders in which values may be provided (or "bound") at a later time after compilation. The following statements are examples of parameterized queries:

```
insert into foods (id, name) values (?,?);
insert into episodes (id, name) (:id, :name);
```

These statements represent two forms of parameter binding: *positional* and *named*. The first command uses positional parameters, and the second command uses named parameters.

Positional parameters are defined by the position of the question mark in the statement. The first question mark has position 1, the second 2, and so on. Named parameters use actual variable names, which are prefixed with a colon. When `sqlite3_prepare_v2()` compiles a statement with parameters, it allocates placeholders for the parameters in the resulting statement handle. It then expects values to be

provided for these parameters before the statement is executed. If you don't bind a value to a parameter, SQLite will use `NULL` as the default when it executes the statement.

The advantage of parameter binding is that you can execute the same statement multiple times without having to recompile it. You just reset the statement, bind a new set of values, and reexecute. This is where resetting rather than finalizing a statement comes in handy: it avoids the overhead of SQL compilation. By resetting a statement, you are reusing the compiled SQL code. You completely avoid the tokenizing, parsing, and code generation overhead. Resetting a statement is implemented in the API by the `sqlite3_reset()` function.

The other advantage of parameters is that SQLite takes care of escaping the string values you bind to parameters. For example, if you had a parameter value such as `'Kenny's Chicken'`, the parameter binding process will automatically convert it to `'Kenny''s Chicken'`—escaping the single quote for you, helping you avoid syntax errors and possible SQL injection attacks (covered in the section "Formatting SQL Statements"). The following pseudocode illustrates the basic process of using bound parameters:

```
db = open('foods.db')
stmt = db.prepare('insert into episodes (id, name) values (:id, :name)')

stmt.bind('id', '1')
stmt.bind('name', 'Soup Nazi')
stmt.step()

# Reset and use again
stmt.reset()
stmt.bind('id', '2')
stmt.bind('name', 'The Junior Mint')

# Done
stmt.finalize()

db.close()
```

Here, `reset()` simply deallocates the statement's resources but leaves its VDBE byte code and parameters intact. The statement is ready to run again without the need for another call to `prepare()`. This can significantly improve performance of repetitive queries such as this because the compiler completely drops out of the equation.

Executing Wrapped Queries

As mentioned earlier, there are two very useful utility functions that wrap the prepared query process, allowing you to execute SQL commands in a single function call. One function—`sqlite3_exec()`—is typically for queries that don't return data. The other—`sqlite3_get_table()`—is typically for queries that do. In many language extensions, you will see analogs to both functions. Most extensions refer to the first method simply as `exec()` and the second as just `get_table()`.

The `exec()` function is a quick and easy way to execute `insert`, `update`, and `delete` statements or DDL statements for creating and destroying database objects. It works straight from the database connection, taking a `sqlite3` handle to an open database along with a string containing one or more SQL statements. That's right; `exec()` is capable of processing a string of *multiple* SQL statements delimited by semicolons and running them all together.

Internally, `exec()` parses the SQL string, identifies individual statements, and then processes them one by one. It allocates its own statement handles and prepares, executes, and finalizes each statement.

If multiple statements are passed to it and one of them fails, `exec()` terminates execution on that command, returning the associated error code. Otherwise, it returns a success code. The following pseudocode illustrates conceptually how `exec()` works in an extension:

```
db = open('foods.db')
db.exec("insert into episodes (id, name) values (1, 'Soup Nazi')")
db.exec("insert into episodes (id, name) values (2, 'The Fusilli Jerry')")
db.exec("begin; delete from episodes; rollback")
db.close()
```

Although you can also use `exec()` to process records returned from `SELECT`, it involves subtle methods for doing so that are generally supported only by the C API.

The second query function, `sqlite3_get_table()`, is somewhat of a misnomer because it is not restricted to just querying a single table. Rather, its name refers to the tabular results of a `select` query. You can certainly process joins with it just as well. In many respects, `get_table()` works in the same way as `exec()`, but it returns a complete result set in memory. This result set is represented in various ways depending on the extension. The following pseudocode illustrates how it is typically used:

```
db = open('foods.db')
table = db.get_table("select * from episodes limit 10")

for i=0; i < table.rows; i++
    for j=0; j < table.cols; j++
        print table[i][j]
    end
end

db.close()
```

The upside of `get_table()` is that it provides a one-step method to query and get results. The downside is that it stores the results completely in memory. So, the larger the result set, the more memory it consumes. Not surprisingly, then, it is not a good idea to use `get_table()` to retrieve large result sets. The prepared query method, on the other hand, holds only one record (actually its associated database page) in memory at a time, so it is much better suited for traversing large result sets.

Notice that although these functions buy you convenience, you also lose a bit of control simply by not having access to a statement handle. For example, you can't use parameterized SQL statements with either of them. So, they are not going to be as efficient for repetitive tasks that could benefit from parameters. Also, the API includes functions that work with statement handles that provide lots of information about columns in a result set—both data and metadata. These are not available with wrapped queries either. Wrapped queries have their uses, to be sure. But prepared queries do as well.

Handling Errors

The previous examples are greatly oversimplified to illustrate the basic parts of query processing. In real life, you always have to consider the possibility of errors. Almost every function you have seen so far can encounter errors of some sort. Common error codes you need to be prepared to handle include `SQLITE_ERROR and SQLITE_BUSY`. The latter error code refers to busy conditions that arise when a connection can't get a lock. Busy conditions are addressed in the "Transactions" section, while schema errors are covered in detail in Chapter 6.

With regard to general errors, the API provides the return code of the last-called function with the `sqlite3_errcode()` function. You can get more specific error information using the `sqlite3_errmsg()` function, which provides a text description of the last error. Most language extensions support this function in some way or another.

With this in mind, each call in the previous example should check for the appropriate errors using something like the following:

```
# Check and report errors
if db.errcode() != SQLITE_OK
    print db.errmsg(stmt)
end
```

In general, error handling is not difficult. The way you handle any error depends on what exactly you are trying to do. The easiest way to approach error handling is the same as with any other API—read the documentation on the function you are using and code defensively.

Formatting SQL Statements

Another nice convenience function you may see some extensions support is `sqlite3_mprintf()`. It is a variant of the standard C library `sprintf()`. It has special substitutions that are specific to SQL that can be very handy. These substitutions are denoted `%q` and `%Q`, and they escape SQL-specific values. `%q` works like `%s` in that it substitutes a null-terminated string from the argument list. But it also doubles every single-quote character, making your life easier and helping guard against SQL injection attacks (see the sidebar "SQL Injection Attacks"). Here's an example:

```
char* before = "Hey, at least %q no pig-man.";
char* after = sqlite3_mprintf(before, "he's");
```

The value `after` produced here is `'Hey, at least he''s no pig-man'`. The single quote in *he's* is doubled, making it acceptable as a string literal in a SQL statement. The `%Q` formatting does everything `%q` does, but it additionally encloses the resulting string in single quotes. Furthermore, if the argument for `%Q` is a `NULL` pointer (in C), it produces the string `NULL` without single quotes. For more information, see the `sqlite3_mprintf()` documentation in the C API reference in Appendix B.

SQL INJECTION ATTACKS

If your application relies on any user input with which to construct SQL statements, you could be vulnerable to a SQL injection attack. If you are not careful to filter user input, it could be possible for someone to craft input that could alter the SQL statement, injecting a new SQL statement into the string. For example, say your program uses user input to fill in the value of the following SQL statement:

```
select * from foods where name='%s';
```

You replace the `%s` with whatever the user supplies. If users have any knowledge of your database, they could provide input that can dramatically alter the SQL statement. For example, say the user were to provide the following string value for the `name` input:

```
nothing' limit 0; select name from sqlite_master where name='%
```

After substituting the user's input into your SQL statement, the new statement turns into two statements:

```
select * from foods where name='nothing' limit 0; select name from
sqlite_master where name='%';
```

The first statement will return nothing, and the second will return the names of all objects in your database. Granted, the odds of this happening require quite a bit of knowledge on the attacker's part, but it is nevertheless possible. Some major (commercial) web applications have been known to keep SQL statements embedded in their JavaScript, which can provide plenty of hints about the database being used. In the previous example, all a malicious user has to do now is insert `drop table` statements for every table found in `sqlite_master`, and you could find yourself fumbling through backups.

Operational Control

The API includes a variety of commands that allow you to monitor, control, or generally limit what can happen in a database. SQLite implements them in the form of filters or callback functions that you can register to be called for specific events. There are three "hook" functions: `sqlite3_commit_hook()`, which monitors transaction commits on a connection; `sqlite3_rollback_hook()`, which monitors rollbacks; and `sqlite3_update_hook()`, which monitors changes to rows from `insert`, `update`, and `delete` operations.

■ **Note** The forthcoming release of SQLite 3.7 will include many new features, one of which is the Write Ahead Log (WAL). With this, a further type of hook named `wal_hook()` will be implemented. This is a little different to the hooks discussed here. More details on WAL will be covered in Chapter 11.

These hooks are called at runtime—while a command is executed. Each hook allows you to register a callback function on a connection-by-connection basis and lets you provide some kind of application-specific data to be passed to the callback as well. The general use of operational control functions is as follows:

```
def commit_hook(cnx)
  log('Attempted commit on connection %x', cnx)
  return -1
end
db = open('foods.db')
db.set_commit_hook(rollback_hook, cnx)
db.exec("begin; delete from episodes; rollback")
db.close()
```

A hook's return value has the power to alter the event in specific ways, depending on the hook. In this example, because the commit hook returns a nonzero value, the commit will be rolled back.

Additionally, the API provides a very powerful compile time hook called `sqlite3_set_authorizer()`. This function provides you with fine-grained control over almost everything that happens in the database as well as the ability to limit both access and modification on a database, table, and column basis. This function is covered in detail in Chapter 6.

Using Threads

SQLite has a number of functions for using it in a multithreaded environment. With version 3.3.1, SQLite introduced a unique operational mode called *shared cache mode*, which is designed for multithreaded embedded servers. This model provides a way for a single thread to host multiple connections that share a common page cache, thus lowering the overall memory footprint of the server. It also employs a different concurrency model. Included with this feature are various functions for managing memory and fine-tuning the server. This operational mode is explained further later in the section "Shared Cache Mode" and in full detail in Chapter 6.

The Extension API

The extension API in the SQLite C API offers support for user-defined functions, aggregates, and collations. A user-defined function is a SQL function that maps to some handler function that you implement in C or another language. When using the C API, you implement this handler in C or C++. In language extensions, you implement the handler in the same language as the extension.

User-defined extensions must be registered on a connection-by-connection basis as they are stored in program memory. That is, they are not stored in the database, like stored procedures in larger relational database systems. They are stored in your program. When your program or script starts up, it is responsible for registering the desired user-defined extensions for each connection that it intends to use them.

Creating User-Defined Functions

Implementing a user-defined function is a two-step process. First, you write the handler. The handler does something that you want to perform from SQL. Next, you register the handler, providing its SQL name, its number of arguments, and a pointer (or reference) to the handler.

For example, say you wanted to create a special SQL function called `hello_newman()`, which returns the text `'Hello Jerry'`. In the SQLite C API, you would first create a C function to implement this, such as the following:

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    /* Create Newman's reply */
    const char *msg = "Hello Jerry";

    /* Set the return value.*/
    sqlite3_result_text(ctx, msg, strlen(msg), SQLITE_STATIC);
}
```

Don't worry if you don't know C or the C API. This handler just returns `'Hello Jerry'`. Next, to actually use it, you register this handler using the `sqlite3_create_function()` (or the equivalent function in your language):

```
sqlite3_create_function(db, "hello_newman", 0, hello_newman);
```

The first argument (`db`) is the database connection. The second argument is the name of the function as it will appear in SQL, and the third argument means that the function takes zero arguments. (If you provide -1, then it means that the function accepts a variable number of arguments.) The last argument is a pointer to the `hello_newman()` C function, which will be called when the SQL function is called.

Once registered, SQLite knows that when it encounters the SQL function `hello_newman()`, it needs to call the C function `hello_newman()` to obtain the result. Now, you can execute `select` hello_newman() within your program, and it will return a single row with one column containing the text `'Hello Jerry'`. As mentioned in Chapter 4, user-defined functions are a handy way to implement specialized domain constraints by embedding them in `check` constraints.

Creating User-Defined Aggregates

Aggregate functions are functions that are applied to all records in a result set and compute some kind of aggregate value from them. `sum()`, `count()`, and `avg()` are examples of standard SQL aggregate functions in SQLite.

Implementing user-defined aggregates is a three-step process in which you register the aggregate, implement a step function to be called for each record in the result set, and implement a finalize function to be called after record processing. The `finalize` function allows you to compute the final aggregate value and do any necessary cleanup.

The following is an example of implementing a simple `SUM()` aggregate called `pysum` in one of the SQLite Python extensions:

```
connection=apsw.Connection("foods.db")

def step(context, *args):
    context['value'] += args[0]

def finalize(context):
    return context['value']

def pysum():
    return ({'value' : 0}, step, finalize)

connection.createaggregatefunction("pysum", pysum)

c = connection.cursor()
print c.execute("select pysum(id) from foods").next()[0]
```

The `createaggregatefunction()` function registers the aggregate, passing in the `step()` function and the `finalize` function. SQLite passes `step()` a context, which it uses to store the intermediate value between calls to `step()`. In this case, it is the running sum. SQLite calls `finalize()` after it has processed the last record. Here, `finalize()` just returns the aggregated sum. SQLite automatically takes care of cleaning up the context.

Creating User-Defined Collations

Collations define how string values are compared. User-defined collations therefore provide a way to create different text comparison and sorting methods. This is done in the API by the `sqlite3_create_collation()` function. SQLite provides three default collations: `BINARY`, `NOCASE`, and `RTRIM`. `BINARY` compares string values using the C function `memcmp()` (which for all intents and purposes is case sensitive). `NOCASE` is just the opposite—its sorting is case insensitive. The `RTRIM` collation works like BINARY except that it ignores trailing spaces.

User-defined collations are especially helpful for locales that are not well served by the default `BINARY` collation or those that need support for UTF-16. They can also be helpful in specific applications such as sorting date formats that don't lend themselves to both lexicographical and chronological order. Chapter 7 illustrates implementing a user-defined collation to sort Oracle dates natively in SQLite.

Transactions

By now you should have a picture of how the API is laid out. You've seen different ways to execute SQL commands along with some helpful utility functions. Executing SQL commands, however, involves more than just knowing what's in the API. Transactions and locks are closely intertwined with query processing. Queries are always performed within transactions, transactions involve locks, and locks can cause problems if not managed properly. You can control both the type and duration of locks by how you use SQL and the way you write code.

Chapter 4 illustrated a specific scenario where deadlocks can arise just by the way that two connections manage transactions through SQL alone. As a programmer, you will have another variable to juggle—code—which can contain multiple connections in multiple states with multiple statement handles on multiple tables at any given time. All it takes is a single statement handle, and your code may be holding an `EXCLUSIVE` lock without you even realizing it, preventing other connections from getting anything done.

That is why it is critical that you have good grasp of how both transactions and locks work and how they relate to the various API functions used to perform queries. Ideally, you should be able to look at the code you write and tell what transaction states it will be in or at least be able to spot potential problems. In this section, we will explore the mechanics behind transactions and locks and in the next section observe them at work in actual code.

Transaction Life Cycles

There are a couple of things to consider with code and transactions. First there is the issue of knowing which objects run under which transactions. Next there is the question of duration—when does a transaction start and when does it end, and at what point does it start to affect other connections? The first question relates directly to the API. The second relates to SQL in general and SQLite's implementation in particular.

As you know, multiple statement handles can be allocated from a single connection. As shown in Figure 5-2, each connection has exactly one B-tree and one pager object associated with it per database. The pager plays a bigger role than the connection in this discussion because it manages transactions, locks, the memory cache, and crash recovery—all of which will be covered in the next few sections. You could just as easily say that the connection object handles all of this, but in reality it is the pager within it. The important thing to remember is that when you write to the database, you do so with one connection, one transaction at a time. Thus, all statement objects run within the single transaction context of the connections from which they are derived. This answers the first question.

As for the second question, transaction duration (or the transaction life cycle) is as short as a single statement, or as long as you like—until you say stop. By default, a connection operates in autocommit mode, which means that every command you issue runs under a separate transaction. Conversely, when you issue a `begin`, the transaction endures until you call either a `COMMIT` or a `rollback`, or until one of your SQL commands causes a constraint violation that results in a `rollback`. The next question is how transactions relate to locks.

Lock States

For the most part, lock duration shadows transaction duration. Although the two don't always start together, they do always finish together. When you conclude a transaction, you free its associated lock. A better way of saying this is that a lock is never released until its associated transaction concludes or, in the worse case, the program crashes. And if the program or system crashes, the transaction doesn't conclude in which case there is still an implicit lock on the database that will be resolved by the next connection to access it. This is covered later in the sidebar "Locking and Crash Recovery."

There are five different locks states in SQLite, and a connection is always in one of them no matter what it's doing. Figure 5-3 shows SQLite's lock states and transitions. This diagram details every possible lock state a connection can be in as well as every path it can take through the life of a transaction. The diagram is represented in terms of lock states, lock transitions, and transaction life cycles. What you are really looking at in the figure are the lives of transactions in terms of locks.

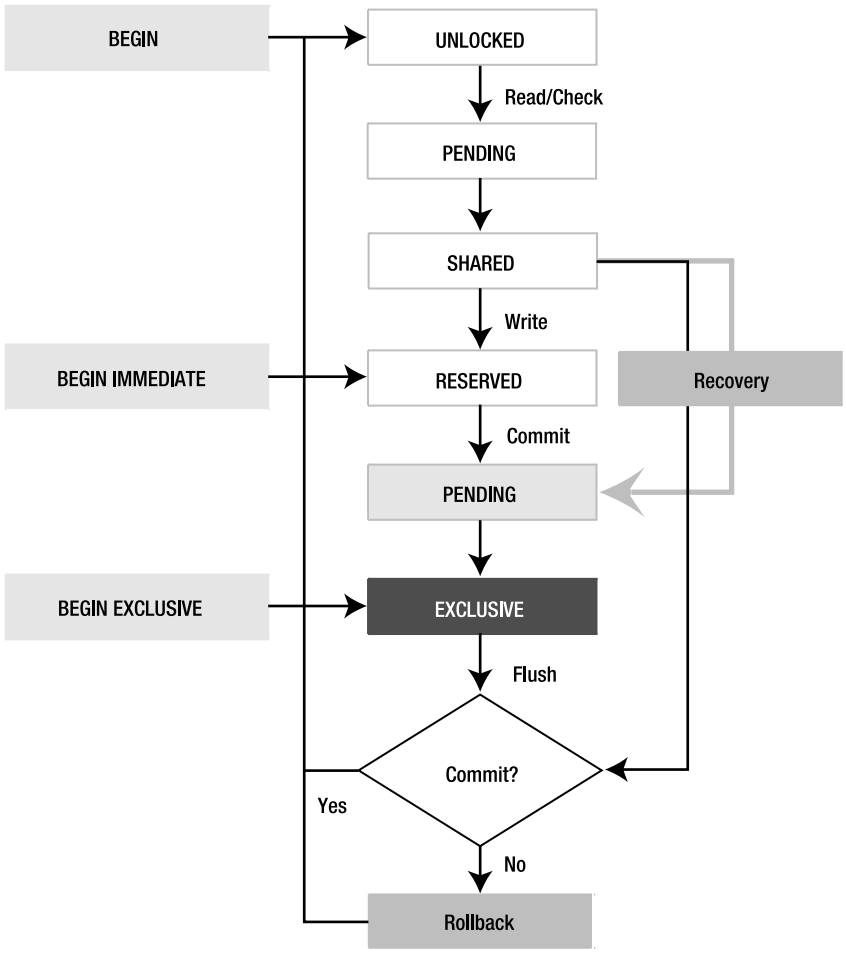


Figure 5-3. *SQLite lock transitions*

Each state has a corresponding lock with the exception of `UNLOCKED`. So, you can say a connection "has a `RESERVED` lock" or "is in the `RESERVED` state" or just "is in `RESERVED`," and it all means the same thing. With the exception of `UNLOCKED`, for a connection to be in a given state it must first obtain the associated lock.

Every transaction starts at `UNLOCKED`, `RESERVED`, or `EXCLUSIVE`. By default, everything begins in `UNLOCKED`, as you can see in Figure 5-3. The locks states in white—`UNLOCKED`, `PENDING`, `SHARED`, and `RESERVED`—can all exist at the same time between connections in a database. Starting with `PENDING` in gray, however, things become more restrictive. The gray `PENDING` state represents the lock being held by a single connection, namely, a writer that wants to get to `EXCLUSIVE`. Conversely, the white `PENDING` state represents a path where connections acquire and release the lock on their way to `SHARED`. Despite all of these different lock states, every transaction in SQLite boils down to one of two types: transactions that

Download from Wow! eBook <www.wowebook.com>

read and transactions that write. That ultimately is what paints the locking picture: readers versus writers and how they get along with each other.

Read Transactions

To start with, let's go through the lock process of a `select` statement. Its path is very simple. A connection that executes a `select` statement starts a transaction, which goes from `UNLOCKED` to `SHARED` and upon `commit` back to `UNLOCKED`. End of story.

Now, out of curiosity, what happens when you execute two statements? What is the lock path then? Well, it depends on whether you are running in autocommit or not. Consider the following example:

```
db = open('foods.db')
db.exec('begin')
db.exec('select * from episodes')
db.exec('select * from episodes')
db.exec('commit')
db.close()
```

Here, with an explicit `begin`, the two `select` commands execute within a single transaction and therefore are executed in the same `SHARED` state. The first `exec()` runs, leaving the connection in `SHARED`, and then the second `exec()` runs. Finally, the manual `commit` takes the connection from `SHARED` back to `UNLOCKED`. The code's lock path would be as follows:

```
UNLOCKED→PENDING →SHARED →UNLOCKED
```

Now consider the case where there are no `begin` and `commit` lines in the example. Then the two `select` commands run in autocommit mode. They will therefore go through the entire path independently. The lock path for the code now would be as follows:

```
UNLOCKED →PENDING →SHARED →UNLOCKED →PENDING →SHARED →UNLOCKED
```

Since the code is just reading data, it may not make much of a difference, but it does have to go through twice the file locks in autocommit mode than it does otherwise. And, as you will see, a writer could sneak in between the two `select` commands and modify the database between `exec()` calls, so you can't be sure that the two commands will return the same results. With `begin..commit`, on the other hand, they are guaranteed to be identical in their results.

Write Transactions

Now let's consider a statement that writes to the database, such as an `update`. First, the connection has to follow the same path as `select` and get to `SHARED`. Every operation—read or write—has to start by going through `UNLOCKED →PENDING →SHARED`, `PENDING`, as you will soon see, is a gateway lock.

The Reserved State

The moment the connection tries to write anything to the database, it has to go from `SHARED` to `RESERVED`. If it gets the `RESERVED` lock, then it is ready to start making modifications. Even though the connection

cannot actually modify the database at this point, it can store modifications in a localized memory cache inside the pager, called the *page cache,* mentioned earlier. This cache is the same cache you configure with the `cache_size` pragma, as described in Chapter 4.

When the connection enters `RESERVED`, the pager initializes the *rollback journal*. This is a file (shown in Figure 5-1) that is used in rollbacks and crash recovery. Specifically, it holds the database pages needed to restore the database to its original state before the transaction. These database pages are put there by the pager when the B-tree modifies a page. In this example, for every record the `update` command modifies, the pager takes the database page associated with the *original* record and copies it out to the journal. The journal then holds some of the contents of the database before the transaction. Therefore, all the pager has to do in order to undo any transaction is to simply copy the contents in the journal back into the database file. Then the database is restored to its state before the transaction.

In the `RESERVED` state, there are actually three sets of pages that the pager manages: modified pages, unmodified pages, and journal pages. Modified pages are just that—pages containing records that the B-tree has changed. These are stored in the page cache. Unmodified pages are pages the B-tree read but did not change. These are a product of commands such as `select`. Finally, there are journal pages, which are the original versions of modified pages. These are not stored in the page cache but rather written to the journal before the B-tree modifies a page.

Because of the page cache, a writing connection can indeed get real work done in the `RESERVED` state without interfering with other (reading) connections. Thus, SQLite can effectively have multiple readers and one writer both working on the same database at the same time. The only catch is that the writing connection has to store its modifications in its page cache, not in the database file. Note also that there can be only one connection in `RESERVED` or `EXCLUSIVE` for a given database at a given time—multiple readers but only one writer.

The Pending State

When the connection finishes making changes for the `update` and the time comes to commit the transaction, the pager begins the process of entering the `EXCLUSIVE` state. You already know how this works from Chapter 4, but we will repeat it for the sake of completeness. From the `RESERVED` state, the pager tries to get a `PENDING` lock. Once it does, it holds onto it, preventing any other connections from getting a `PENDING` lock. Look at Figure 5-3 and see what effect this has. Remember we told you that `PENDING` was a gateway lock. Now you see why. Since the writer is holding onto the `PENDING` lock, nobody else can get to `SHARED` from `UNLOCKED` anymore. The result is that no new connections can enter the database: no new readers, no new writers. This `PENDING` state is the attrition phase. The writer is guaranteed that it can wait in line for the database and—as long as everyone behaves properly—get it. Only other sessions that already have `SHARED` locks can continue to work as normal. In `PENDING`, the writer waits for these connections to finish and release their locks. What's involved with waiting for locks is a separate issue, which will be addressed shortly in the section "Waiting for Locks."

When the other connections release their locks, the database then belongs to the writer. Then, the pager moves from `PENDING` to `EXCLUSIVE`.

The Exclusive State

During `EXCLUSIVE`, the main job is to flush the modified pages from the page cache to the database file. This is when things get serious, because the pager is going to actually modify the database. It goes about this with extreme caution.

Before the pager begins writing the modified pages, it first tends to the journal. It checks that the complete contents of the journal have been written to disk. At this point, it is very likely that even though the pager has written pages to the journal file, the operating system has buffered many if not all of them

in memory. The pager tells the operating system to literally write all of these pages to the disk. This is where the synchronous pragma comes into play, as described in Chapter 4. The method specified by synchronous determines how careful the pager is to ensure that the operating system commits journal pages to disk. The normal setting is to perform a single "sync" before continuing, telling the operating system to confirm that all buffered journal pages are written to the disk surface. If `synchronous` is set to `FULL`, then the pager does two full "syncs" before proceeding. If `synchronous` is set to `NONE`, the pager doesn't bother with the journal at all (and while it can be 50 times faster, you can kiss transaction durability goodbye).

The reason that committing the journal to disk is so important is that if the program or system crashes while the pager is writing to the database file, the journal is the only way to restore the database file later. If the journal's pages weren't completely written to disk before a system crash, then the database cannot be restored fully to its original state, because the journal pages that were in memory were lost in the crash. In this case, you have an inconsistent database at best and a corrupted one at worse.

■ **Caution** Even if you use the most conservative setting for the synchronous pragma, you still may not be guaranteed that the journal is truly committed to disk. This is no fault of SQLite, but rather of certain types of hardware and operating systems. SQLite uses the `fsync()` system call on Unix and `FlushFileBuffers()` on Windows to force journal pages to disk. But it has been reported that these functions don't always work, especially with cheap IDE disks. Apparently, some manufacturers of IDE disks use controller chips that tend to bend the truth about actually committing data to disk. In some cases, the chips cache the data in volatile memory while reporting that they wrote it to the drive surface. Also, there have been (unconfirmed) reports that Windows occasionally ignores `FlushFileBuffers()`. If you have hardware or software that lies to you, your transactions may not be as durable as you might think.

Once the journal is taken care of, the pager then copies all of the modified pages to the database file. What happens next depends on the transaction mode. If, as in this case, the transaction autocommits, then the pager cleans up the journal, clears the page cache, and proceeds from `EXCLUSIVE` to `UNLOCKED`. If the transaction does not commit, then the pager continues to hold the `EXCLUSIVE` lock, and the journal stays in play until either a `COMMIT` or `ROLLBACK` is issued.

Autocommit and Efficiency

With all this in mind, consider what happens with `UPDATE`s that run in an explicit transaction versus ones that run in autocommit. In autocommit, every command that modifies the database runs in a separate transaction and travels through the following path:

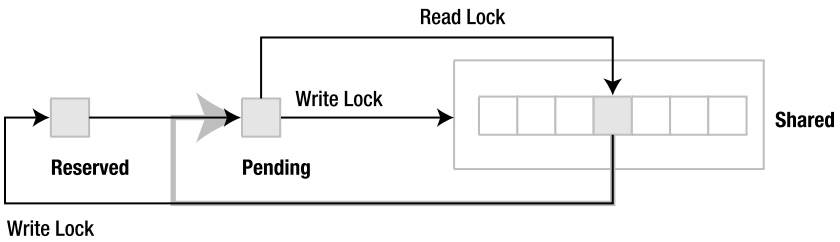
```
UNLOCKED →PENDING →SHARED →RESERVED →PENDING →EXCLUSIVE →UNLOCKED
```

Additionally, each trip through this path involves creating, committing, and clearing out the rollback journal. This all adds up. Although running multiple `SELECT`s in autocommit mode is perhaps not that big of a deal performance-wise, you should really rethink autocommit mode for frequent writes. And, as mentioned in the `SELECT` case, when running multiple commands in autocommit, there is

nothing stopping another connection from coming along and modifying the database in between commands. If you have two updates that depend on specific values in the database, you should always run them in the same transaction for this reason.

LOCKING AND CRASH RECOVERY

SQLite's lock implementation is based on standard file locking. SQLite keeps three different file locks on the database file: a reserved byte, a pending byte, and a shared region.



Everything starts at the pending byte. To move from `UNLOCKED` to `SHARED`, a connection first attempts to get a read-lock on the pending byte. If successful, it gets a read lock on a random byte in the shared region and releases the read lock on the pending byte. To move from `SHARED` to `RESERVED`, a connection attempts to obtain a write lock on the reserved byte. To get from `RESERVED` to `EXCLUSIVE`, a connection attempts to get a write lock on the pending byte. If successful, this is what causes the attrition process, because it is no longer possible for other connections to get a read lock on the pending byte to enter `SHARED`. Finally, to get the `EXCLUSIVE` lock, the connection attempts to get a write lock on the entire shared region. Since the shared region holds read locks by all other active connections, this step guarantees that an `EXCLUSIVE` lock is granted only after all other `SHARED` locks are first released.

SQLite's crash recovery mechanism uses the reserved byte to determine when a database needs to be restored. Since the journal file and the `RESERVED` lock go hand in hand, if the pager sees the former without the latter, then something is wrong. Every time the pager opens a database file or tries to fetch a page from the database, it does a simple consistency check. If it finds a journal file but no `RESERVED` lock on the database, then the process that created the journal file must have crashed, or the system went down. In this case, the journal is called a hot journal, and the database is potentially in an inconsistent state. To make things right, the journal needs to be "played back" to restore the database to its original state before the interrupted transaction.

To start the play back, the pager puts the database into recovery mode. To do this, it goes directly from `SHARED` to `PENDING` as shown by the gray line in the illustration (and also in Figure 5-3). This is the only time it ever makes this transition. The reason it skips the reserved lock is twofold. First, by locking the pending byte, it keeps all new connections out of the database. Second, connections that are already in the database (in `SHARED`) will also see the hot journal the next time they try to access a page. Those connections will attempt to go into recovery mode and replay the journal as well. However, they won't be able to because the first connection already has the `PENDING` lock. Thus, by going straight from `SHARED` to `PENDING`, the first connection ensures that (1) no new connections can enter the database and (2) active

connections in SHARED cannot go into recovery mode. Everyone but the restoring connection is temporarily suspended.

Basically, a hot journal is an implicit EXCLUSIVE lock. If a writer crashes, no further activity can transpire in the database until some connection restores it. The next pager to access a page will see the hot journal, lock everyone out, and start recovery. If there are no other active connections, then the first program to connect to the database will detect the hot journal and start recovery.

Tuning the Page Cache

If we change the previous example and say that it started with a `begin`, followed by the `update`, and in the middle of making all those modifications the page cache fills up (runs out of memory), how should SQLite respond? That is, the `update` results in more modified pages than will fit in the page cache. What happens now?

Transitioning to Exclusive

The real question is, when exactly does the pager move from RESERVED to EXCLUSIVE and why? There are two scenarios, and you've just seen them both. Either the connection reaches the commit point and deliberately enters EXCLUSIVE or the page cache fills up and it has no other option. We just looked at the commit point scenario. So, what happens when the page cache fills up? Put simply, the pager can no longer store any more modified pages and therefore can no longer do any work. It is forced to move into EXCLUSIVE in order to continue. In reality, this is not entirely true because there is a soft limit and a hard limit.

The soft limit corresponds to the first time the page cache fills up. At this point, the cache is a mixed bag of modified and unmodified pages. In this case, the pager tries to clean out the page cache. It goes through the cache page by page looking for unmodified pages and clearing them out. Once it does that, it can sputter along with what memory has freed up until the cache fills up again. It repeats the process until the cache is completely made up of modified pages. And that is that hard limit. At this point, the pager has no other recourse but to proceed into EXCLUSIVE.

The RESERVED state, then, is where the `cache_size` pragma makes a difference. Just as explained in Chapter 4, `cache_size` controls the size of the page cache. The bigger the page cache, the more modified pages the pager can store, and the more work the connection can do before having to enter EXCLUSIVE. Also, as mentioned, by doing all the database work in RESERVED, you minimize the time in EXCLUSIVE. If you get all your work done in RESERVED, then EXCLUSIVE is held only long enough to flush modified pages to disk—not compile *more* queries and process *more* results and then write to disk. Doing the processing in RESERVED can significantly increase overall concurrency. Ideally, if you have a large transaction or a congested database and you can spare the memory, try to ensure that your cache size is large enough to hold your connection in RESERVED as long as possible.

Sizing the Page Cache

So, how do you determine what the cache size should be? It depends on what you are doing. Say that you want to update every record in the `episodes` table. In this case, you know that every page in the table will be modified. Therefore, you figure out how many pages are in `episodes` and adjust the cache size accordingly. You can get all the information you need on `episodes` using `sqlite3_analyzer`. For each

table, it will dump detailed statistics, including the total page count. For example, if you run it on the `foods` database, you get the following information about `episodes`:

```
*** Table EPISODES ***************************************************

Percentage of total database.....  20.0%
Number of entries..... 181
Bytes of storage consumed..... 5120
Bytes of payload..... 3229        63.1%
Average payload per entry..... 17.84
Average unused bytes per entry........ 5.79
Average fanout..... 4.00
Maximum payload per entry..... 38
Entries that use overflow..... 0            0.0%
Index pages used..... 1
Primary pages used..... 4
Overflow pages used..... 0
Total pages used..... 5
Unused bytes on index pages..... 990         96.7%
Unused bytes on primary pages..... 58         1.4%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 1048        20.5%
```

The total page count is 5. But of those, only 4 pages of actual table are used—1 page is an index. Since the default cache size is 2,000 pages, you have nothing to worry about. There are about 400 records in `episodes`, which means there are about 100 records per page. You wouldn't have to worry about adjusting the page cache before updating every record unless there were at least 196,000 rows in `episodes`. And remember, you would only need to do this in environments where there are other connections using the database and concurrency is an issue. If you are the only one using the database, then it really wouldn't matter.

Waiting for Locks

We talked earlier about the pager waiting to go from `PENDING` to `EXCLUSIVE`. What exactly is involved with waiting on a lock? First, any call to `exec()` or `step()` can involve waiting on a lock. Whenever SQLite encounters a situation where it can't get a lock, the default behavior is to return `Delete SQLITE_BUSY` to the function that caused it to seek the lock. Regardless of the command you execute, you can potentially encounter `SQLITE_BUSY`. `SELECT` commands, as you know by now, can fail to get a `SHARED` lock if a writer is pending or writing. The simple thing to do when you get `SQLITE_BUSY` is to just retry the call. However, we will see shortly that this is not always the best course of action.

Using a Busy Handler

Instead of just retrying the call over and over, you can use a *busy handler*. Rather than having the API return `SQLITE_BUSY` if a connection cannot get a lock, you can get it to call the busy handler instead.

A busy handler is a function you create that kills time or does whatever else you want it to do—it can send spam to your mother-in-law for all SQLite cares. It's just going to get called when SQLite can't get a lock. The only thing the busy handler *has* to do is provide a return value, telling SQLite what to do next.

By convention, if the handler returns true, then SQLite will continue to try for the lock. If it returns false, SQLite will then return `SQLITE_BUSY` to the function requesting the lock. Consider the following example:

```
counter = 1

def busy()
    counter = counter + 1
    if counter == 2
        return 0
    end

    spam_mother_in_law(100)
    return 1
end

db.busy_handler(busy)
stmt = db.prepare('select * from episodes;')
stmt.step()
stmt.finalize()
```

The implementation of `spam_mother_in_law()` is left as an exercise for the reader.

The `step()` function has to get a `SHARED` lock on the database to perform the `select`. However, say there is a writer active. Normally, `step()` would return `SQLITE_BUSY`. However, in this case it doesn't. The pager (which is one that deals with locks) calls the `busy()` function instead, because it has been registered as the busy handler. `busy()` increments a counter, forwards your mother-in-law 100 random messages from your spam folder, and returns 1, which the pager interprets as true—keep trying to get the lock. The pager then tries again to get the `SHARED` lock. Say the database is still locked. The pager calls the busy handler again. Only this time, `busy()` returns 0, which the pager interprets as false. In this case, rather than retrying the lock, the pager sends `SQLITE_BUSY` up the stack, and that's what `step()` ends up returning.

If you just want to kill time waiting for a lock, you don't have to write your own busy handler. The SQLite API has one for you. It is a simple busy handler that sleeps for a given period of time waiting for a lock. In the API, the function is called `sqlite3_busy_timeout()`, and it is supported by some extension libraries. You can essentially say "try sleeping for 10 seconds when you can't get a lock," and the pager will do that for you. If it sleeps for 10 seconds and still can't get the lock, then it will return `SQLITE_BUSY`.

Using the Right Transaction

Let's consider the previous example again, but this time the command is an `update` rather than a `select`. What does `SQLITE_BUSY` actually mean now? In `select`, it just means, "I can't get a `SHARED` lock." But what does is mean for an `update`? The truth is, you don't really know what it means. `SQLITE_BUSY` could mean that the connection failed to get a `SHARED` lock because there is a writer pending. It could also mean that it got a `SHARED` lock but couldn't get to `RESERVED`. The point is you don't know the state of the database or the state of your connection for that matter. In autocommit mode, `SQLITE_BUSY` for a write operation is completely indeterminate. So, what do you do next? Should you just keep calling `step()` over and over until the command goes through?

Here's the thing to think about. Suppose `SQLITE_BUSY` was the result of you getting a `SHARED` lock but not `RESERVED`, and now you are holding up a connection in `RESERVED` from getting to `EXCLUSIVE`. Again, *you don't know the state of the database.* And just using a brute-force method to push your transaction through is not necessarily going to work, for you or any other connection. If you just keep calling `step()`,

you are just going to butt heads with the connection that has the `RESERVED` lock, and if neither of you backs down, you'll deadlock.

■ **Note** SQLite tries to help with deadlock prevention in this particular scenario by ignoring the busy handler of the offending connection. The `SHARED` connection's busy handler will not be invoked if it is preventing a `RESERVED` connection from proceeding. However, it is up to the code to get the hint. The code can still just repeatedly call `step()` over and over, in which case there is nothing more SQLite can do to help.

Since you know you want to write to the database, then you need to start by issuing `begin IMMEDIATE`. If you get a `SQLITE_BUSY`, then at least you know what state you're in. You know you can safely keep trying without holding up another connection. And once you finally do succeed, you know what state you are in then as well—`RESERVED`. Now you can use brute force if you have to because you are the one in the right. If you start with a `begin exclusive`, on the other hand, then you are assured that you won't have any busy conditions to deal with at all. Just remember that in this case you are doing your work in `EXCLUSIVE`, which is not as good for concurrency as doing the work in `RESERVED`.

LOCKS AND NETWORK FILE SYSTEMS

At this point, you should have a good appreciation for what can go wrong when a database is shared over a network file system. SQLite handles concurrency by placing file locks on the database file. It is very important that these locks be both set and released at the right times. SQLite is completely dependent on the file system to manage locks correctly for concurrent use. SQLite uses the same locking mechanisms regardless of whether it is running on a normal file system or network file system. It uses POSIX advisory locks on Unix, Linux, and OS X, and the `LockFile()`, `LockFileEx()`, and `UnlockFile()` system calls on Windows. These calls are standard system calls and work correctly on normal file systems. It is the network file system's job to emulate a normal file system. And unfortunately, this doesn't always work correctly with some implementations. And even if the network file system works correctly, there still other things to consider.

Take NFS, for example. It's a great network file system. However, many original NFS implementations were known to have buggy or in some cases unimplemented locking. And with a SQLite database, this can cause serious problems. Without locking, two connections can get an `EXCLUSIVE` lock on the same database and write to it at the same time, leading to an almost certain database corruption. This is not a problem with the NFS protocol in general, but with some implementations of it in particular. Thankfully, most modern implementations of NFS have overcome these issues, and some implementations—such as those from Sun—are now rock-solid.

NFS can still have issues with locking if one of the clients goes down (taking their database lock with it) and does not come back up. For NFSv3, a locked file would need administrative intervention to clear it. For NFSv4, there would be a timeout period. But how long is the timeout? Thirty seconds? A minute? The real issue is not NFS but specific applications of it. Network file systems, even if perfectly implemented, are not necessarily going to work exactly like a local file system.

> The bottom line is, if you are going to mix concurrency and network file systems, there are many issues to consider, even if you are using the best network file system out there. It's not as simple as running on a local file system.

Code

By now, you have a pretty good picture of the API, transactions, and locks. To finish up, let's put all three of these things together in the context of your code and consider a few scenarios for which you might want to watch.

Using Multiple Connections

If you have written code that uses other relational databases, you may have written code that used multiple connections within a single block of code. The classic example is having one connection iterate over a table while another connection modifies records in place.

In SQLite, using multiple connections in the same block of code can cause problems. You have to be careful of how you go about it. Consider the following example:

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
  c2.exec('update episodes set <some columns, criteria, etc.>)
end

stmt.finalize()

c1.close()
c2.close()
```

We bet you can easily spot the problem here. In the `while` loop, `c2` attempts an `update`, while `c1` has a `SHARED` lock open. That `SHARED` lock won't be released until `stmt` is finalized after the `while` loop. Therefore, it is impossible to write to the database within the `while` loop. Either the updates will silently fail, or if you have a busy handler, then it will only delay the program. The best thing to do here is to use one connection for the job and to run it under a single `begin immediate` transaction. The new version might be as follows:

```
c1 = open('foods.db')

# Keep trying until we get it
while c1.exec('begin immediate') != SQLITE_SUCCESS
end

stmt = c1.prepare('select * from episodes')
```

```
while stmt.step()
  print stmt.column('name')
  c1.exec('update episodes set <some columns, criteria, etc.>)
end

stmt.finalize()
c1.exec('commit')
c1.close()
```

In cases like this, you should use statements from a single connection for reading or writing. Then you won't have to worry about database locks causing problems. However, as it turns out, this particular example still won't work. If you are iterating over a table with one statement and updating it with another, there is an additional locking issue that you need to know about as well, which we'll cover next.

The Importance of Finalizing

A common gotcha in processing `select` statements is the failure to realize that the `SHARED` lock is not necessarily released until `finalize()` (or `reset()`) is called—well, most of the time. Consider the following example:

```
stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
end

c2.exec('begin immediate; update episodes set …; commit;')

stmt.finalize()
```

Although you should never do this in practice, you might end up doing it anyway by accident simply because you can get away with it. If you write the equivalent of this program in the C API, it will actually work. Even though `finalize()` has not been called, the second connection can modify the database without any problem. Before we tell you why, take a look at the next example:

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('select * from episodes')

stmt.step()
stmt.step()
stmt.step()

c2.exec('begin immediate; update episodes set …; commit;')

stmt.finalize()
```

Download from Wow! eBook <www.wowebook.com>

Let's say that `episodes` has 100 records. And the program stepped through only three of them. What happens here? The second connection will get `SQLITE_BUSY`.

In the first example, SQLite released the `SHARED` lock when the statement reached the end of the result set. That is, in the final call to `step()`, where the API returns `SQLITE_DONE`, the VDBE encountered the `Close` instruction, and SQLite closed the cursor and dropped the `SHARED` lock. Thus, `c2` was able to push its `insert` through even though `c1` had not called `finalize()`.

In the second case, the statement had not reached the end of the set. The next call to `step()` would have returned `SQLITE_RESULT`, which means there are more rows in the results set and that the `SHARED` lock is still active. Thus, `c2` could not get the `insert` through this time because of the `SHARED` lock from `c1`.

The moral of the story is don't do this, even though sometimes you can. Always call `finalize()` or `reset()` before you write with another connection. The other thing to remember is that in autocommit mode `step()` and `finalize()` are more or less transaction and lock boundaries. They start and end transactions. They acquire and release locks. You should be very careful about what you do with other connections in between these functions.

Shared Cache Mode

Now that you are clear on the concurrency rules, it's time for some new confusion. SQLite offers an alternative concurrency model called *shared cache mode,* which relates to how connections can operate within individual threads.

In shared cache mode, a thread can create multiple connections that share the same page cache. Furthermore, this group of connections can have multiple readers *and a single writer* (in `EXCLUSIVE`) working on the same database at the same time. The catch is that these connections cannot be shared across threads—they are strictly limited to the thread (running specifically in shared cache mode) that created them. Furthermore, writers and readers have to be prepared to handle a special condition involving table locks.

When readers read tables, SQLite automatically puts table locks on them. This prevents writers from modifying those tables. If a writer tries to modify a read-locked table, it will get `SQLITE_LOCKED`. The same logic applies to readers trying to read from write-locked tables. However, in this latter case, readers can still go ahead and read tables that are being modified by a writer if they run in *read-uncommitted mode,* which is enabled by the `read_uncommitted` pragma. In this case, SQLite does not place read locks on the tables read by these readers. As a result, these readers don't interfere with writers at all. However, these readers can get inconsistent query results, because a writer can modify tables as the readers read them. This is similar to the "dirty read" concept you might have seen in other relational databases.

Shared cache mode is designed for embedded servers that need to conserve memory and have slightly higher concurrency under certain conditions. More information on using it with the C API can be found in Chapter 6.

Summary

The SQLite API is flexible, intuitive, and easy to use. It has two basic parts: the core API and the extension API. The core API revolves around two basic data structures used to execute SQL commands: the connection and the statement. Commands are executed in three steps: compilation, execution, and finalization. SQLite's wrapper functions `exec()` and `get_table()` wrap these steps into a single function call, automatically handling the associated statement object for you. The extension API provides you with the means to customize SQLite in three difference ways: user-defined functions, user-defined aggregates, and user-defined collations.

Because SQLite's concurrency model is somewhat different from other databases, it is important that you understand a bit about how it manages transactions and locks, how they work behind the scenes, and how they work within your code. Overall, the concepts are not difficult to understand, and there are just a few simple rules to keep in mind when you write code that uses SQLite.

The following chapters will draw heavily on what you've have learned here, because these concepts apply not only to the C API but to language extensions as well as they are built on top of the C API.

CHAPTER 6

The Core C API

This chapter covers the part of the SQLite API used to work with databases. You already saw an overview of how the API works in Chapter 5. Now let's concentrate on the specifics.

Starting with a few easily understood examples, we will take an in-depth tour through the C API and expand upon the examples, filling in various details with a variety of useful functions. As we go along, you should see the C equivalents of the model all fall into place, with some additional features you may not have seen before—features that primarily exist only in the C API. By the time we reach the end of this chapter, you should have a good feel for all of the API functions related to running commands, managing transactions, fetching records, handling errors, and performing many other tasks related to general database work.

The SQLite version 3 API consists of dozens and dozens of functions. Only about eight functions, however, are needed to actually connect, process queries, and disconnect from a database. The remaining functions can be arranged into small groups that specialize in accomplishing specific tasks.

Although it is best to read the chapter straight through, if at any point you want more detailed information on a particular function, you can consult the C API reference at `www.sqlite.org/c3ref/intro.html`.

You can find all the examples in this chapter in self-contained example programs in the examples zip file, available on the Apress web site at `www.apress.com`. For every example presented, we specifically identify the name of the corresponding source file from which the example was taken.

Wrapped Queries

You are already familiar with the way that SQLite executes queries, as well as its various wrapper functions for executing SQL commands in a single function call. We will start with the C API versions of these wrappers because they are simple, self-contained, and easy to use. They are a good starting point, which will let you have some fun and not get too bogged down with details. Along the way, I'll introduce some other handy functions that go hand in hand with query processing. By the end of this section, you will be able connect, disconnect, and query a database using the wrapped queries.

Connecting and Disconnecting

Before you can execute SQL commands, you first have to connect to a database. Connecting to a database is perhaps best described as *opening* a database, because SQLite databases are contained in single operating system files (one file to one database). Similarly, the preferred term for disconnecting is *closing* the database.

You open a database with the `sqlite3_open_v2()`, `sqlite3_open()`, or `sqlite3_open16()` functions, which have the following declaration(s):

```
int sqlite3_open_v2(
  const char *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
  int flags,              /* Flags */
  const char *zVfs        /* Name of VFS module to use */
);

int sqlite3_open (
  const void *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);

int sqlite3_open16(
  const void *filename,   /* Database filename (UTF-16) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);
```

Typically, you'll use `sqlite3_open_v2()`, because this is the latest and greatest function in SQLite for opening your database, and it accommodates more options and capabilities over the old `sqlite3_open()`. Regardless of the function chosen, the `filename` argument can be the name of an operating system file, the text string `':memory:'`, or an empty string or a `NULL` pointer. If you use `':memory:'`, `sqlite3_open_v2()` will create an in-memory database in RAM that lasts only for the duration of the session. If `filename` is an empty string or a `NULL`, `sqlite3_open_v2()` opens a temporary disk file that is automatically deleted when the connection closes. Otherwise, `sqlite3_open_v2()` attempts to open the database file by using its value. If no file by that name exists, `sqlite3_open_v2()` will open a new database file by that name if the `SQLITE_OPEN_CREATE` flag is included in the third parameter, or it will return an error if the `SQLITE_OPEN_CREATE` flag is omitted.

■ **Note** Here we have included both the UTF-8 and UTF-16 declarations for `sqlite3_open()`, as well as the more modern `sqlite3_open_v2()`. From here on out, we will refer to the UTF-8 declarations only for the sake of brevity, and where they exist, we'll deal with the latest function signatures that incorporate SQLite's latest functionality, rather than cover all the legacy functions as well. Therefore, please keep in mind that there are many functions in the API that have UTF-16 forms, as well as older UTF-8 forms for backward compatibility. The complete C API reference is available on the SQLite web site at `www.sqlite.org/c3ref/intro.html`.

The `flags` parameter is a bit vector that can include the following values: `SQLITE_OPEN_READONLY`, `SQLITE_OPEN_READWRITE`, and `SQLITE_OPEN_CREATE`. The names are reasonable self-explanatory, but a few subtleties are worth noting. `SQLITE_OPEN_READONLY` and `SQLITE_OPEN_READWRITE` open a SQLite database in read-only or read/write mode as their names suggest. Both options require that the database file already exist; otherwise, an error will be returned. `SQLITE_OPEN_CREATE` combined with `SQLITE_OPEN_READWRITE` exhibits the behavior we've seen in the first five chapters, which is the legacy `sqlite3_open()` behavior. Where a database already exists, it is opened for reading and writing. If the

database specified doesn't exist, it is created (though as we've pointed out quite a few times, the act of persisting the database to disk will be pending the creation of the first object).

The `flags` parameter can also be combined with the `SQLITE_OPEN_NOMUTEX`, `SQLITE_OPEN_FULLMUTEX`, `SQLITE_OPEN_SHAREDCACHE`, or `SQLITE_OPEN_PRIVATECACHE` flags to further control transactional behavior for the database handle. The final parameter, `zVfs`, allows the caller to override the default `sqlite3_vfs` operating system interface.

Upon completion, `sqlite3_open_v2()` will initialize the `sqlite3` structure passed into it by the `ppDb` argument. This structure should be considered as an opaque handle representing a single connection to a database. This is more of a connection handle than a database handle since it is possible to attach multiple databases to a single connection. However, this connection still represents exactly one transaction context regardless of how many databases are attached.

You close a connection by using the `sqlite3_close()` function, which is declared as follows:

```
int sqlite3_close(sqlite3*);
```

For `sqlite3_close()` to complete successfully, all prepared queries associated with the connection must be finalized. If any queries remain that have not been finalized, `sqlite3_close()` will return `SQLITE_BUSY` with the error message "Unable to close due to unfinalized statements."

■ **Note** If there is a transaction open on a connection when it is closed by `sqlite3_close()`, the transaction will automatically be rolled back.

The exec Query

The `sqlite3_exec()` function provides a quick, easy way to execute SQL commands and is especially handy for commands that modify the database (that is, don't return any data). This is often also referred to as a *convenience* function, which nicely wraps up a lot of other tasks in one easy API call. This function has the following declaration:

```
int sqlite3_exec(
  sqlite3*,                      /* An open database */
  const char *sql,               /* SQL to be executed */
  sqlite_callback,               /* Callback function */
  void *data                     /* 1st argument to callback function */
  char **errmsg                  /* Error msg written here */
);
```

The SQL provided in the `sql` argument can consist of more than one SQL command. `sqlite3_exec()` will parse and execute every command in the `sql` string until it reaches the end of the string or encounters an error. Listing 6-1 (taken from `create.c`) illustrates how to use `sqlite3_exec()`.The example opens a database called test.db and creates within it a single table called `episodes`. After that, it inserts one record. The `create table` command will physically create the database file if it does not already exist.

Listing 6-1. *Using* `sqlite3_exec()` *for Simple Commands*

```
int main(int argc, char **argv)
{
    sqlite3 *db;
    char *zErr;
    int rc;
    char *sql;

    rc = sqlite3_open_v2("test.db", &db);

    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    sql = "create table episodes(id int, name text)";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);

    if(rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }

    sql = "insert into episodes values (10, 'The Dinner Party')";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);

    sqlite3_close(db);
    return 0;
}
```

As mentioned in Chapter 5, it is actually possible to get records from `sqlite3_exec()`, although you don't see it implemented much outside of the C API. `sqlite3_exec()` contains a callback mechanism that provides a way to obtain results from `select` statements. This mechanism is implemented by the third and fourth arguments of the function. The third argument is a pointer to a callback function. If it's provided, SQLite will call the function for each record processed in each `select` statement executed within the `sql` argument. The callback function has the following declaration:

```
typedef int (*sqlite3_callback)(
    void*,    /* Data provided in the 4th argument of sqlite3_exec() */
    int,      /* The number of columns in row                         */
    char**,   /* An array of strings representing fields in the row  */
    char**    /* An array of strings representing column names       */
);
```

The fourth argument to `sqlite3_exec()` is a void pointer to any application-specific data you want to supply to the callback function. SQLite will pass this data as the first argument of the callback function.

The final argument (`errmsg`) is a pointer to a string to which an error message can be written should an error occur during processing. Thus, `sqlite3_exec()` has two sources of error information. The first is the return value. The other is the human-readable string, assigned to `errmsg`. If you pass in a `NULL` for `errmsg`, then SQLite will not provide any error message. Note that if you do provide a pointer for `errmsg`, the memory used to create the message is allocated on the heap. You should therefore check for a non-`NULL` value after the call and use `sqlite3_free()` to free the memory used to hold the `errmsg` string if an error occurs.

■ **Note** Note that you can pass a NULL into `sqlite3_free()`. The result is a harmless no-op. So, you can, if you want, call `sqlite3_free(errmsg)` without having to check to see whether `errmsg` is not `NULL`. It mainly depends on where and how you want to interrogate any actual errors and respond accordingly.

Putting it all together, `sqlite3_exec()` allows you to issue a batch of commands, and you can collect all the returned data by using the callback interface. For example, let's insert a record into the `episodes` table and then select all of its records, all in a single call to `sqlite3_exec()`. The complete code, shown in Listing 6-2, is taken from `exec.c`.

Listing 6-2. *Using `sqlite3_exec()` for Record Processing*

```
int callback(void* data, int ncols, char** values, char** headers);

int main(int argc, char **argv)
{
    sqlite3 *db;
    int rc;
    char *sql;
    char *zErr;

    rc = sqlite3_open_v2("test.db", &db);

    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    const char* data = "Callback function called";
    sql = "insert into episodes (id, name) values (11,'Mackinaw Peaches');"
          "select * from episodes;";
    rc = sqlite3_exec(db, sql, callback, data, &zErr);
```

```
    if(rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }

    sqlite3_close(db);
    return 0;
}

int callback(void* data, int ncols, char** values, char** headers)
{
    int i;
    fprintf(stderr, "%s: ", (const char*)data);
    for(i=0; i < ncols; i++) {
        fprintf(stderr, "%s=%s ", headers[i], values[i]);
    }

    fprintf(stderr, "\n");
    return 0;
}
```

SQLite parses the `sql` string; runs the first command, which inserts a record; and then runs the second command, consisting of the `select` statement. For the second command, SQLite calls the callback function for each record returned. Running the program produces the following output:

```
    Callback function called: id=10 name=The Dinner Party
    Callback function called: id=11 name=Mackinaw Peaches
```

Notice that the callback function returns 0. This return value actually exerts some control over `sqlite3_exec()`. If the callback function returns a nonzero value, then `sqlite3_exec()` will abort (in other words, it will terminate all processing of this and subsequent commands in the `sql` string).

So, `sqlite3_exec()` provides an easy way to modify the database and also provides an interface with which to process records. Why then should you bother with prepared queries? Well, as you will see in the next section, there are quite a few advantages:

- Prepared queries don't require a callback interface, which makes coding simple and more linear.
- Prepared queries have associated functions that provide better column information. You can obtain a column's storage type, declared type, schema name (if it is aliased), table name, and database name. `sqlite3_exec()`'s callback interface provides just the column names.
- Prepared queries provide a way to obtain field/column values in other data types besides text and in native C data types such as `int` and `double`, whereas `sqlite3_exec()`'s callback interface only provides fields as string values.

- Prepared queries can be rerun, allowing you to reuse the compiled SQL.
- Prepared queries support parameterized SQL statements.

EXAMINING CHANGES

If you are performing an `update` or a `delete`, you may want to know how many records were affected. You can get this information from `sqlite3_changes()`. It provides the number of affected records for the last executed `update`, `insert`, or `delete` statement. Obviously, if you are running a batch of queries (multiple commands in the `sql` argument of `sqlite3_exec()`), this function will be good only for the last command in the batch. Note also that `sqlite3_changes()` does not include changes activated from the result of triggers fired from your original command.

If you are performing an insert on a table with an autoincrement column, odds are you will eventually want to know the primary key value of an inserted record. In such cases, you can use `sqlite3_last_insert_rowid()` to obtain this value. You can also do this from within SQL as well via the `last_insert_rowid()` SQL function.

The Get Table Query

The `sqlite3_get_table()` function returns an entire result set of a command in a single function call. Just as `sqlite3_exec()` wraps the prepared query API functions, allowing you to run them all at once, `sqlite3_get_table()` wraps `sqlite3_exec()` for commands that return data with just as much convenience. Using `sqlite3_get_table()`, you don't have to bother with the `sqlite3_exec()` callback function, thus making it easier to fetch records. `sqlite3_get_table()` has the following declaration:

```
int sqlite3_get_table(
  sqlite3*,              /* An open database */
  const char *sql,       /* SQL to be executed */
  char ***resultp,       /* Result written to a char *[] that this points to */
  int *nrow,             /* Number of result rows written here */
  int *ncolumn,          /* Number of result columns written here */
  char **errmsg          /* Error msg written here */
);
```

This function takes all the records returned from the SQL statement in `sql` and stores them in the `resultp` argument using memory declared on the heap (using sqlite3_`malloc()`). The memory must be freed using `sqlite3_free_table()`, which takes the `resultp` pointer as its sole argument. The first record in `resultp` is actually not a record but contains the names of the columns in the result set. Examine the code fragment in Listing 6-3 (taken from `get_table.c`).

Listing 6-3. *Using* `sqlite3_get_table`

```
int main(int argc, char **argv)
{
    /* Connect to database, etc. */

    char *result[];
    sql = "select * from episodes;";
    rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);

    /* Do something with data */

    /* Free memory */
    sqlite3_free_table(result)
}
```

If, for example, the result set returned is of the following form:

```
name               | id
-----------------------
The Junior Mint    | 43
The Smelly Car     | 28
The Fusilli Jerry  | 21
```

then the format of the result array will be structured as follows:

```
result [0] = "name";
result [1] = "id";
result [2] = "The Junior Mint";
result [3] = "43";
result [4] = "The Smelly Car";
result [5] = "28";
result [6] = "The Fusilli Jerry";
result [7] = "21";
```

The first two elements contain the column headings of the result set. Therefore, you can think of the result set indexing as 1-based with respect to rows but 0-based with respect to columns. An example may help clarify this. Listing 6-4 shows the code to print out each column of each row in the result set.

Listing 6-4. *Iterating Through* `sqlite3_get_table()` *Results*

```
rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);

for(i=0; i < nrows; i++) {
    for(j=0; j < ncols; j++) {
        /* the i+1 term skips over the first record,
        which is the column headers */
        fprintf(stdout, "%s", result[(i+1)*ncols + j]);
    }
}
```

Download from Wow! eBook <www.wowebook.com>

Prepared Queries

Our Chapter 5 overview of SQLite's API introduced the concepts of the prepare, step, and finalize functions. This section covers all aspects of this process, including stepping through result sets, fetching records, and using parameterized queries. We told you prepared statements were infinitely preferable to the convenience wrapper functions, and this is where we prove the point.

The wrapper functions simply wrap all these steps into a single function call, making it more convenient in some situations to run specific commands. Each query function provides its own way of getting at rows and columns. As a general rule, the more packaged the method is, the less control you have over execution and results. Therefore, prepared queries offer the most features, the most control, and the most information, with `sqlite3_exec()` offering slightly fewer features and `sqlite3_get_table()` offering fewer still.

Prepared queries use a special group of functions to access field and column information from a row. You get column values using `sqlite3_column_xxx()`, where *xxx* represents the data type of the value to be returned (for example, `int`, `double`, `blob`). You can retrieve data in whatever format you like. You can also obtain the declared types of columns (as they are defined in the `create table` statement) and other miscellaneous metadata such as storage format and both associated table and database names. `sqlite3_exec()`, by comparison, provides only a fraction of this information through its callback function. The same is true with `sqlite3_get_table()`, which only includes the result set's column headers with the data.

In practice, you will find that each query method has its uses. `sqlite3_exec()` is especially good for running commands that modify the database (`create`, `drop`, `insert`, `update`, and `delete`). One function call, and it's done. Prepared queries are typically better for `select` statements because they offer so much more information, more linear coding (no callback functions), and more control by using cursors to iterate over results.

As you'll recall from Chapter 5, prepared queries are performed in three basic steps: compilation, execution, and finalization. You compile the query with `sqlite3_prepare_v2()`, execute it step-by-step using `sqlite3_step()`, and close it using `sqlite3_finalize()`, or you can reuse it using `sqlite3_reset()`. This process and the individual steps are all explained in detail in the following sections.

Compilation

Compilation, or preparation, takes a SQL statement and compiles it into byte code readable by the virtual database engine (VDBE). It is performed by `sqlite3_prepare_v2()`, which is declared as follows:

```
int sqlite3_prepare_v2(
  sqlite3 *db,            /* Database handle          */
  const char *zSql,       /* SQL text, UTF-8 encoded  */
  int nBytes,             /* Length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle    */
  const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
```

`sqlite3_prepare_v2()` compiles the first SQL statement in the `zSQL` string (which can contain multiple SQL statements). It allocates all the resources necessary to execute the statement and associates it along with the byte code into a single statement handle (also referred to as simply a *statement*), designated by the out parameter `ppStmt`, which is a `sqlite3_stmt` structure. From the programmer's perspective, this structure is little more than an opaque handle used to execute a SQL statement and obtain its associated records. However, this data structure contains the command's byte

code, bound parameters, B-tree cursors, execution context, and any other data `sqlite3_step()` needs to manage the state of the query during execution.

`sqlite3_prepare_v2()` does not affect the connection or database in any way. It does not start a transaction or get a lock. It works directly with the compiler, which simply prepares the query for execution. Statement handles are highly dependent on the database schema with which they were compiled. If another connection alters the database schema, between the time you prepare a statement and the time you actually run it, your prepared statement will expire. However, `sqlite3_prepare_v2()` is built to automatically attempt to recompile (re-prepare) your statement if possible and will do this silently if a schema change has invalidated your existing statement. If the schema has changed in such a way as to make recompilation impossible, your call to `sqlite3_step()` with the statement will lead to a `SQLITE_SCHEMA` error, which is discussed later in the section "Errors and the Unexpected." At this point, you would need to examine the error associated with `SQLITE_SCHEMA` by using the `sqlite3_errmsg()` function.

Execution

Once you prepare the query, the next step is to execute it using `sqlite3_step()`, declared as follows:

```
int sqlite3_step(sqlite3_stmt *pStmt);
```

`sqlite3_step()` takes the statement handle and talks directly to the VDBE, which steps through its byte-code instructions one by one to carry out the SQL statement. On the first call to `sqlite3_step()`, the VDBE obtains the requisite database lock needed to perform the command. If it can't get the lock, `sqlite3_step()` will return `SQLITE_BUSY`, if there is no busy handler installed. If one is installed, it will call that handler instead.

For SQL statements that don't return data, the first call to `sqlite3_step()` executes the command in its entirety, returning a result code indicating the outcome. For SQL statements that do return data, such as `select`, the first call to `sqlite3_step()` positions the statement's B-tree cursor on the first record. Subsequent calls to `sqlite3_step()` position the cursor on subsequent records in the result set. `sqlite3_step()` returns `SQLITE_ROW` for each record in the set until it reaches the end, whereupon it returns `SQLITE_DONE`, indicating that the cursor has reached the end of the set.

■ **Note** For those of you familiar with older versions of SQLite, these result codes are part of the more advanced—and some would say more correct—set of result codes that replace the legacy simple return values like `SQLITE_ERROR`. A full list of result codes and extended result codes is available at `www.sqlite.org/c3ref/c_abort.html`

All other API functions related to data access use the statement's cursor to obtain information about the current record. For example, the `sqlite3_column_xxx()` functions all use the statement handle, specifically its cursor, to fetch the current record's fields.

Finalization and Reset

Once the statement has reached the end of execution, it must be finalized. You can either finalize or reset the statement using one of the following functions:

```
int sqlite3_finalize(sqlite3_stmt *pStmt);
int sqlite3_reset(sqlite3_stmt *pStmt);
```

`sqlite3_finalize()` will close out the statement. It frees resources and commits or rolls back any implicit transactions (if the connection is in autocommit mode), clearing the journal file and freeing the associated lock.

If you want to reuse the statement, you can do so using `sqlite3_reset()`. It will keep the compiled SQL statement (and any bound parameters) but commits any changes related to the current statement to the database. It also releases its lock and clears the journal file if autocommit is enabled. The primary difference between `sqlite3_finalize()` and `sqlite3_reset()` is that the latter preserves the resources associated with the statement so that it can be executed again, avoiding the need to call `sqlite3_prepare()` to compile the SQL command.

Let's go through an example. Listing 6-5 shows a simple, complete program using a prepared query. It is taken from `select.c` in the examples.

Listing 6-5. *Using Prepared Queries*

```
int main(int argc, char **argv)
{
    int rc, i, ncols;
    sqlite3 *db;
    sqlite3_stmt *stmt;
    char *sql;
    const char *tail;

    rc = sqlite3_open_v2("foods.db", &db);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    sql = "select * from episodes;";

    rc = sqlite3_prepare_v2(db, sql, -1, &stmt, &tail);

    if(rc != SQLITE_OK) {
        fprintf(stderr, "SQL error: %s\n", sqlite3_errmsg(db));
    }

    rc = sqlite3_step(stmt);
    ncols = sqlite3_column_count(stmt);
```

```
    while(rc == SQLITE_ROW) {

        for(i=0; i < ncols; i++) {
            fprintf(stderr, "'%s' ", sqlite3_column_text(stmt, i));
        }

        fprintf(stderr, "\n");

        rc = sqlite3_step(stmt);
    }

    sqlite3_finalize(stmt);
    sqlite3_close(db);

    return 0;
}
```

This example connects to the `foods.db` database, queries the `episodes` table, and prints out all columns of all records within it. Keep in mind this is a simplified example—there are a few other things we need to check for when calling `sqlite3_step()`, such as errors and busy conditions, but we will address them later.

Like `sqlite3_exec()`, `sqlite3_prepare_v2()` can accept a string containing multiple SQL statements. However, unlike `sqlite3_exec()`, it will process only the first statement in the string. But it does make it easy for you to process subsequent SQL statements in the string by providing the `pzTail` out parameter. After you call `sqlite3_prepare()`, it will point this parameter (if provided) to the starting position of the next statement in the `zSQL` string. Using `pzTail`, processing a batch of SQL commands in a given string can be executed in a loop as follows:

```
while(sqlite3_complete(sql) ){
    rc = sqlite3_prepare(db, sql, -1, &stmt, &tail);

    /* Process query results */

    /* Skip to next command in string. */
    sql = tail;
}
```

This example uses another API function not yet covered—`sqlite3_complete()`, which does as its name suggests. It takes a string and returns true if there is at least one complete (but not necessarily valid) SQL statement in it, and it returns false otherwise. In reality, `sqlite_complete()` looks for a semicolon terminator for the string (and accounting for literals in the SQL). So, although its name suggests some kind of infallible observer checking your statements, it's really just a handy tool for things such as showing you the continuation prompt in the `sqlite` command line when writing multiline statements and doing other similar useful tasks.

Fetching Records

So far, you have seen how to obtain records and columns from `sqlite3_exec()` and `sqlite3_get_table()`. Prepared queries, by comparison, offer many more options when it comes to getting information from records in the database.

For a statement that returns records, the number of columns in the result set can be obtained using `sqlite3_column_count()` and `sqlite3_data_count()`, which are declared as follows:

```
int sqlite3_column_count(sqlite3_stmt *pStmt);
int sqlite3_data_count(sqlite3_stmt *pStmt);
```

`sqlite3_column_count()` returns the number of columns associated with a statement handle. You can call it on a statement handle before it is actually executed. If the query in question is not a `select` statement, `sqlite3_column_count()` will return 0. Similarly, `sqlite3_data_count()` returns the number of columns for the current record, after `sqlite3_step()` returns `SQLITE_ROW`. This function will work only if the statement handle has an active cursor.

Getting Column Information

You can obtain the name of each column in the current record using `sqlite3_column_name()`, which is declared as follows:

```
const char *sqlite3_column_name( sqlite3_stmt*, /* statement handle */
                                 int iCol       /* column ordinal   */);
```

Similarly, you can get the associated storage class for each column using `sqlite3_column_type()`, which is declared as follows:

```
int sqlite3_column_type( sqlite3_stmt*, /* statement handle */
                         int iCol       /* column ordinal   */);
```

This function returns an integer value that corresponds to one of five storage class codes, defined as follows:

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_TEXT     3
#define SQLITE_BLOB     4
#define SQLITE_NULL     5<A NAME="50520099_sqlite3_column_name16">
```

These are SQLite's native data types, or storage classes, as described in Chapter 4. All data stored within a SQLite database is stored in one of these five forms, depending on its initial representation and the affinity of the column. For our purposes, the terms *storage class* and *data type* are synonymous. For more information on storage classes, see the sections "Storage Classes" and "Type Affinity" in Chapter 4.

You can obtain the declared data type of a column as it is defined in the table's schema using the `sqlite3_column_decltype()` function, which is declared as follows:

```
const char *sqlite3_column_decltype( sqlite3_stmt*, /* statement handle */
                                     int            /* column ordinal   */);
```

If a column in a result set does not correspond to an actual table column (say, for example, the column is the result of a literal value, expression, function, or aggregate), this function will return `NULL` as the declared type of that column. For example, suppose you have a table in your database defined as follows:

```
CREATE TABLE t1(c1 INTEGER);
```

Then you execute the following query:

```
SELECT c1 + 1, 0 FROM t1;
```

In this case, `sqlite3_column_decltype()` will return `INTEGER` for the first column and `NULL` for the second.

In addition to the declared type, you can obtain other information on a column using the following functions:

```
const char *sqlite3_column_database_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_table_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_origin_name(sqlite3_stmt *pStmt, int iCol);
```

The first function will return the database associated with a column, the second will return its table, and the last function will return the column's actual name as defined in the schema. That is, if you assigned the column an alias in the SQL statement, `sqlite3_column_origin_name()` will return its actual name as defined in the schema. Note that these functions are available only if you compile SQLite with the `SQLITE_ENABLE_COLUMN_METADATA` preprocessor directive.

Getting Column Values

You can obtain the values for each column in the current record using the `sqlite3_column_`*`xxx`*`()` functions, which are of the following general form:

```
xxx sqlite3_column_xxx( sqlite3_stmt*, /* statement handle */
                        int iCol       /* column ordinal   */);
```

Here *xxx* is the data type you want the data represented in (for example, `int`, `blob`, `double`, and so on). These are the most commonly used of the `sqlite3_column_`*`xxx`*`()` functions:

```
int sqlite3_column_int(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
```

For each function, SQLite converts the internal representation (storage class in the column) to the type specified in the function name. There are a number of rules SQLite uses to convert the internal data type representation to that of the requested type. Table 6-1 lists these rules.

Table 6-1. *Column Type Conversion Rules*

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0.
NULL	FLOAT	Result is 0.0.
NULL	TEXT	Result is a NULL pointer.
NULL	BLOB	Result is a NULL pointer.
INTEGER	FLOAT	Convert from integer to float.
INTEGER	TEXT	Result is the ASCII rendering of the integer.
INTEGER	BLOB	Result is the ASCII rendering of the integer.
FLOAT	INTEGER	Convert from float to integer.
FLOAT	TEXT	Result is the ASCII rendering of the float.
FLOAT	BLOB	Result is the ASCII rendering of the float.
TEXT	INTEGER	Use atoi().
TEXT	FLOAT	Use atof().
TEXT	BLOB	No change.
BLOB	INTEGER	Convert to TEXT and then use atoi().
BLOB	FLOAT	Convert to TEXT and then use atof().
BLOB	TEXT	Add a \000 terminator if needed.

Like the sqlite3_bind_*xxx*() functions described later, BLOBs require a little more work in that you must specify their length in order to copy them. For BLOB columns, you can get the actual length of the data using sqlite3_column_bytes(), which is declared as follows:

```
int sqlite3_column_bytes( sqlite3_stmt*, /* statement handle   */
                          int            /* column ordinal */);
```

Once you get the length, you can copy the binary data using sqlite3_column_blob(). For example, say the first column in the result set contains binary data. One way to get a copy of that data would be as follows:

```
int len = sqlite3_column_bytes(stmt,0);
void* data = malloc(len);
memcpy(data, len, sqlite3_column_blob(stmt,0));
```

A Practical Example

To help solidify all these column functions, Listing 6-6 (taken from `columns.c`) illustrates using the functions we've described to retrieve column information and values for a simple `select` statement.

Listing 6-6. *Obtaining Column Information*

```
int main(int argc, char **argv)
{
    int rc, i, ncols, id, cid;
    char *name, *sql;
    sqlite3 *db;
    sqlite3_stmt *stmt;

    sql = "select id, name from episodes";
    sqlite3_open_v2("test.db", &db);

    sqlite3_prepare_v2(db, sql, strlen(sql), &stmt, NULL);

    ncols = sqlite3_column_count(stmt);
    rc = sqlite3_step(stmt);

    /* Print column information */
    for(i=0; i < ncols; i++) {
        fprintf(stdout, "Column: name=%s, storage class=%i, declared=%s\n",
                         sqlite3_column_name(stmt, i),
                         sqlite3_column_type(stmt, i),
                         sqlite3_column_decltype(stmt, i));
    }

    fprintf(stdout, "\n");

    while(rc == SQLITE_ROW) {
        id = sqlite3_column_int(stmt, 0);
        cid = sqlite3_column_int(stmt, 1);
        name = sqlite3_column_text(stmt, 2);
        if(name != NULL){
            fprintf(stderr, "Row:  id=%i, cid=%i, name='%s'\n", id,cid,name);
        } else {
            /* Field is NULL */
            fprintf(stderr, "Row:  id=%i, cid=%i, name=NULL\n", id,cid);
        }
        rc = sqlite3_step(stmt);
    }
```

```
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return 0;
}
```

This example connects to the database, selects records from the `episodes` table, and prints the column information and the fields for each row (using their internal storage class). Running the program produces the following output:

```
Column: name=id, storage class=1, declared=integer
Column: name=name, storage class=3, declared=text

Row:  id=1, name='The Dinner Party'
Row:  id=2, name='The Soup Nazi'
Row:  id=3, name='The Fusilli Jerry'
```

FINDING A STATEMENT'S CONNECTION

In practice, you may find yourself writing code where some of your functions have access only to the statement handle, not the connection handle. If these functions encounter an error while working with the statement handle, they will not have a way to get error information from `sqlite3_errmsg()`, because it requires a connection handle to work. This is where `sqlite3_db_handle()` comes in handy. It is declared as follows:

```
int sqlite3_db_handle(sqlite3_stmt*);
```

Given a statement handle, `sqlite3_db_handle()` returns the associated connection handle. This way, you don't need to worry about having to pass the connection handle along with the statement handle everywhere you process query results.

Parameterized Queries

The API includes support for designating parameters in a SQL statement, allowing you to provide (or "bind") values for them at a later time. Bound parameters are used in conjunction with `sqlite3_prepare()`. For example, you could create a SQL statement like the following:

```
insert into foo values (?,?,?);
```

Then you can, for example, bind the integer value 2 to the first parameter (designated by the first `?` character), the string value `'pi'` to the second parameter, and the double value 3.14 for the third parameter, as illustrated in the following code (taken from `parameters.c`):

```
const char* sql = "insert into foo values(?,?,?)";
sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);
```

```
sqlite3_bind_int(stmt,    1, 2);
sqlite3_bind_text(stmt,   2, "pi");
sqlite3_bind_double(stmt, 3, 3.14);

sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

This generates and executes the following statement:

```
insert into foo values (2, 'pi', 3.14)
```

This particular method of binding uses *positional parameters* (as described in Chapter 5) where each parameter is designated by a question mark (?) character and later identified by its index or relative position in the SQL statement.

Before delving into the other parameter methods, it is helpful to first understand the process by which parameters are defined, bound, and evaluated. When you write a parameterized statement such as the following, the parameters within it are identified when `sqlite3_prepare()` compiles the query:

```
insert into episodes (id,name) values (?,?)
```

`sqlite3_prepare()` recognizes that there are parameters in a SQL statement. Internally, it assigns each parameter a number to uniquely identify it. In the case of positional parameters, it starts with 1 for the first parameter found and uses sequential integer values for subsequent parameters. It stores this information in the resulting statement handle (`sqlite3_stmt` structure), which will then expect a specific number of values to be bound to the given parameters before execution. If you do not bind a value to a parameter, `sqlite3_step()` will use `NULL` for its value by default when the statement is executed.

After you prepare the statement, you then bind values to it. You do this using the `sqlite3_bind_`*`xxx`*`()` functions, which have the following general form:

```
sqlite3_bind_xxx( sqlite3_stmt*, /* statement handle  */
                  int i,         /* parameter number  */
                  xxx value      /* value to be bound */
                 );
```

The *xxx* in the function name represents the data type of the value to bind. For example, to bind a `double` value to a parameter, you would use `sqlite3_bind_double()`, which is declared as follows:

```
sqlite3_bind_double(sqlite3_stmt* stmt, int i, double value);
```

The common bind functions are as follows:

```
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int64(sqlite3_stmt*, int, long long int);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
```

Download from Wow! eBook <www.wowebook.com>

In general, the bind functions can be divided into two classes, one for scalar values (`int`, `double`, `int64`, and `NULL`) and the other for arrays (`blob`, `text`, and `text16`). They differ only in that the array bind functions require a length argument and a pointer to a cleanup function. Also, `sqlite3_bind_text()` automatically escapes quote characters like `sqlite3_mprintf()`. Using the `BLOB` variant, the array bind function has the following declaration:

```
int sqlite3_bind_blob( sqlite3_stmt*,   /* statement handle       */
                       int,             /* ordinal               */
                       const void*,     /* pointer to blob data  */
                       int n,           /* length (bytes) of data */
                       void(*)(void*)); /* cleanup hander        */
```

There are two predefined values for the cleanup handler provided by the API that have special meanings, defined as follows:

```
#define SQLITE_STATIC      ((void(*)(void *))0)
#define SQLITE_TRANSIENT   ((void(*)(void *))-1)
```

Each value designates a specific cleanup action. `SQLITE_STATIC` tells the array bind function that the array memory resides in unmanaged space, so SQLite does not attempt to clean it up. `SQLITE_TRANSIENT` tells the bind function that the array memory is subject to change, so SQLite makes its own private copy of the data, which it automatically cleans up when the statement is finalized. The third option is to provide a pointer to your own cleanup function, which must be of the following form:

```
void cleanup_fn(void*)
```

If provided, SQLite will call your cleanup function, passing in the array memory when the statement is finalized.

■ **Note** Bound parameters remain bound throughout the lifetime of the statement handle. They remain bound even after a call to `sqlite3_reset()` and are freed only when the statement is finalized (by calling `sqlite3_finalize()`).

Once you have bound all the parameters you want, you can then execute the statement. You do this using the next function in the sequence: `sqlite3_step()`. `sqlite3_step()` will take the bound values, substitute them for the parameters in the SQL statement, and then begin executing the command.

Now that you understand the binding process, the four parameter-binding methods differ only by the following:

- The way in which parameters are represented in the SQL statement (using a positional parameter, explicitly defined parameter number, or alphanumeric parameter name)
- The way in which parameters are assigned numbers

For positional parameters, `sqlite3_prepare()` assigns numbers using sequential integer values starting with 1 for the first parameter. In the previous example, the first `?` parameter is assigned 1, and the second `?` parameter is assigned 2. With positional parameters, it is your job to keep track of which number corresponds to which parameter (or question mark) in the SQL statement and correctly specify that number in the bind functions.

Numbered Parameters

Numbered parameters, on the other hand, allow you to specify your own numbers for parameters, rather than use an internal sequence. The syntax for numbered parameters uses a question mark followed by the parameter number. Take, for example, the following piece of code (taken from `parameters.c`):

```
name = "Mackinaw Peaches";
sql  = "insert into episodes (id, cid, name) "
       "values (?100,?100,?101)";

rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

if(rc != SQLITE_OK) {
    fprintf(stderr, "sqlite3_prepare() : Error: %s\n", tail);
    return rc;
}

sqlite3_bind_int(stmt, 100, 10);
sqlite3_bind_text(stmt, 101, name, strlen(name), SQLITE_TRANSIENT);
sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

This example uses 100 and 101 for its parameter numbers. Parameter number 100 has the integer value 10 bound to it. Parameter 101 has the string value `'Mackinaw Peaches'` bound to it. Note how numbered parameters come in handy when you need to bind a single value in more than one place in a SQL statement. Consider the values part of the previous SQL statement:

```
    insert into episodes (id, cid, name) values (?100,?100,?101)";
```

Parameter 100 is being used twice—once for `id` and again for `cid`. Thus, numbered parameters save time when you need to use a bound value in more than one place.

■ **Note** When using numbered parameters in a SQL statement, keep in mind that the allowable range consists of the integer values 1–999, and for optimal performance and memory utilization, you should choose smaller numbers.

Named Parameters

The third parameter binding method is using named parameters. Whereas you can assign your own numbers using numbered parameters, you can assign alphanumeric names with named parameters. Likewise, because numbered parameters are prefixed with a question mark (?), you identify named parameters by prefixing a colon (:) or at-sign (@) to the parameter name. Consider the following code snippet (taken from `parameters.c`):

```
name = "Mackinaw Peaches";
sql = "insert into episodes (id, cid, name) values (:cosmo,:cosmo,@newman)";

rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

sqlite3_bind_int( stmt,
                  sqlite3_bind_parameter_index(stmt, ":cosmo"), 10);

sqlite3_bind_text( stmt,
                   sqlite3_bind_parameter_index(stmt, "@newman"),
                   name,
                   strlen(name), SQLITE_TRANSIENT );

sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

This example is identical to the previous example using numbered parameters, except it uses two named parameters called `:cosmo` and `@newman` instead. Like positional parameters, named parameters are automatically assigned numbers by `sqlite3_prepare()`. Although the numbers assigned to each parameter are unknown, you can resolve them using `sqlite3_bind_parameter_index()`, which takes a parameter name and returns the corresponding parameter number. This is the number you use to bind the value to its parameter. All in all, named parameters mainly help with legibility more than anything else.

■ **Note** While the function `sqlite3_bind_parameter_index()` seems to refer to a parameter number as an index, for all intents and purposes the two terms (*number* and *index*) are synonymous.

Tcl Parameters

The final parameter scheme is called Tcl parameters and is specific more to the Tcl extension than it is to the C API. Basically, it works identically to named parameters except that rather than using alphanumeric values for parameter names, it uses Tcl variable names. In the Tcl extension, when the Tcl equivalent of `sqlite3_prepare()` is called, the Tcl extension automatically searches for Tcl variables with the given parameter names in the active Tcl program environment and binds them to their respective parameters. Despite its current application in the Tcl interface, nothing prohibits this same mechanism from being applied to other language interfaces, which can in turn implement the same feature. In this respect, referring to this parameter method solely as Tcl parameters may be a bit of a misnomer. The Tcl extension just happened to be the first application that utilized this method. Basically, the Tcl parameter

syntax does little more than provide an alternate syntax to named parameters—rather than prefixing the parameters with a colon (`:`) or an at-sign (`@`), it uses a dollar sign (`$`).

Errors and the Unexpected

Up to now, we have looked at the API from a rather optimistic viewpoint, as if nothing could ever go wrong. But things do go wrong, and there is part of the API devoted to that. The three things you always have to guard against in your code are errors, busy conditions, and, my personal favorite, schema changes.

Handling Errors

Many of the API functions return integer result codes. That means they can potentially return error codes of some sort. The most common functions to watch are typically the most frequently used, such as `sqlite3_open_v2()`, `sqlite3_prepare_v2()`, and friends, as well as `sqlite3_exec()`. You should always program defensively and review every API function before you use it to ensure that you deal with error conditions that can arise. Of all the error results defined in SQLite, only a fraction of them will really matter to your application in practice. All of the SQLite result codes are listed in Table 6-2. API functions that can return them include the following:

```
sqlite3_bind_xxx()
sqlite3_close()
sqlite3_create_collation()
sqlite3_collation_needed()
sqlite3_create_function()
sqlite3_prepare_v2()
sqlite3_exec()
sqlite3_finalize()
sqlite3_get_table()
sqlite3_open_v2()
sqlite3_reset()
sqlite3_step()
```

You can get extended information on a given error using `sqlite3_errmsg()`, which is declared as follows:

```
const char *sqlite3_errmsg(sqlite3*);
```

It takes a connection handle as its only argument and returns the most recent error resulting from an API call on that connection. If no error has been encountered, it returns the string “not an error.”

Table 6-2. *SQLite Result Codes*

Code	Description
`SQLITE_OK`	The operation was successful.
`SQLITE_ERROR`	General SQL error or missing database. It may be possible to obtain more error information depending on the error condition (`SQLITE_SCHEMA`, for example).
`SQLITE_INTERNAL`	Internal logic error.
`SQLITE_PERM`	Access permission denied.
`SQLITE_ABORT`	A callback routine requested an abort.
`SQLITE_BUSY`	The database file is locked.
`SQLITE_LOCKED`	A table in the database is locked.
`SQLITE_NOMEM`	A call to `malloc()` has failed within a database operation.
`SQLITE_READONLY`	An attempt was made to write to a read-only database.
`SQLITE_INTERRUPT`	Operation was terminated by `sqlite3_interrupt()`.
`SQLITE_IOERR`	Some kind of disk I/O error occurred.
`SQLITE_CORRUPT`	The database disk image is malformed. This will also occur if an attempt is made to open a non-SQLite database file as a SQLite database.
`SQLITE_FULL`	Insertion failed because the database is full. There is no more space on the file system or the database file cannot be expanded.
`SQLITE_CANTOPEN`	SQLite was unable to open the database file.
`SQLITE_PROTOCOL`	The database is locked or there has been a protocol error.
`SQLITE_EMPTY`	(Internal only.) The database table is empty.
`SQLITE_SCHEMA`	The database schema has changed.
`SQLITE_CONSTRAINT`	Abort due to constraint violation.

Continued

Code	Description
SQLITE_MISMATCH	Data type mismatch. An example of this is an attempt to insert noninteger data into a column labeled INTEGER PRIMARY KEY. For most columns, SQLite ignores the data type and allows any kind of data to be stored. But an INTEGER PRIMARY KEY column is allowed to store integer data only.
SQLITE_MISUSE	Library was used incorrectly. This error might occur if one or more of the SQLite API routines is used incorrectly.
SQLITE_NOLFS	Uses OS features not supported on host. This value is returned if the SQLite library was compiled with large file support (LFS) enabled but LFS isn't supported on the host operating system.
SQLITE_AUTH	Authorization denied. This occurs when a callback function installed using sqlite3_set_authorizer() returns SQLITE_DENY.
SQLITE_FORMAT	Auxiliary database format error.
SQLITE_RANGE	Second parameter to sqlite3_bind() out of range.
SQLITE_NOTADB	File opened is not a SQLite database file.
SQLITE_ROW	sqlite3_step() has another row ready.
SQLITE_DONE	sqlite3_step() has finished executing.

Although it is very uncommon outside of embedded systems, one of the most critical errors you can encounter is SQLITE_NOMEM, which means that no memory can be allocated on the heap (for example, malloc() failed). SQLite is quite robust in this regard, recovering gracefully from out-of-memory error conditions. It will continue to work assuming the underlying operating system completes memory allocation system calls like malloc().

Handling Busy Conditions

Two important functions related to processing queries are sqlite3_busy_handler() and sqlite3_busy_timeout(). If your program uses a database on which there are other active connections, odds are it will eventually have to wait for a lock and therefore will have to deal with SQLITE_BUSY. Whenever you call an API function that causes SQLite to seek a lock and SQLite is unable to get it, the function will return SQLITE_BUSY. There are three ways to deal with this:

- Handle SQLITE_BUSY yourself, either by rerunning the statement or by taking some other action
- Have SQLite call a busy handler
- Ask SQLite to wait (block or sleep) for some period of time for the lock to clear

The last option involves using sqlite3_busy_timeout(). This function tells SQLite how long to wait for a lock to clear before returning SQLITE_BUSY. Although it can ultimately result in you still having to handle SQLITE_BUSY, in practice setting this value to a sufficient period of time (say 30 seconds) usually provides enough time for even the most intensive transaction to complete. Nevertheless, you should still have some contingency plan in place to handle SQLITE_BUSY.

User-Defined Busy Handlers

The second option entails using sqlite3_busy_handler(). This function provides a way to call a user-defined function rather than blocking or returning SQLITE_BUSY right away. It's declared as follows:

```
int sqlite3_busy_handler(sqlite3*, int(*)(void*,int), void*);
```

The second argument is a pointer to a function to be called as the busy handler, and the third argument is a pointer to application-specific data to be passed as the first argument to the handler. The second argument to the busy handler is the number of prior calls made to the handler for the same lock.

Such a handler might call sleep() for a period to wait out the lock, or it may send some kind of notification. It may do whatever you like, because it is yours to implement. Be warned, though, that registering a busy handler does not guarantee that it will always be called. As mentioned in Chapter 5, SQLite will forego calling a busy handler for a connection if it perceives a deadlock might result. Specifically, if your connection in SHARED is interfering with another connection in RESERVED, SQLite will not invoke your busy handler, hoping you will take the hint. In this case, you are trying to write to the database from SHARED (starting the transaction with BEGIN) when you really should be starting from RESERVED (starting the transaction with BEGIN IMMEDIATE).

The only restriction on busy handlers is that they may not close the database. Closing the database from within a busy handler can delete critical data structures out from under the executing query and result in crashing your program.

Advice

All things considered, the best route may be to set the timeout to a reasonable value and then take some precautions if and when you receive a SQLITE_BUSY value. In general, if you are going to write to the database, start in RESERVED. If you don't do this, then the next best thing is to install a busy handler, set the timeout to a known value, and if SQLite returns SQLITE_BUSY, check the response time. If the time is less than the busy handler's delay, then SQLite is telling you that your query (and connection) is preventing a writer from proceeding. If you want to write to the database at this point, you should finalize or reset and then reexecute the statement, this time starting with BEGIN IMMEDIATE.

Handling Schema Changes

Whenever a connection changes the database schema, all other prepared statements that were compiled before the change are invalidated. The result is that the first call to sqlite3_step() for such statements

will attempt to recompile the relevant SQL and proceed normally from there if possible. If recompilation is impossible (for example, if an object has been dropped entirely), the `sqlite3_step()` returns `SQLITE_SCHEMA`. From a locking standpoint, the schema change occurs between the time a reader calls `sqlite3_prepare()` to compile a statement and calling `sqlite3_step()` to execute it.

When this happens, the only course of action for you is to handle the change in circumstances and start over. Several events can cause `SQLITE_SCHEMA` errors:

- Detaching databases
- Modifying or installing user-defined functions or aggregates
- Modifying or installing user-defined collations
- Modifying or installing authorization functions
- Vacuuming the database

The reason the `SQLITE_SCHEMA` condition exists ultimately relates to the VDBE. When a connection changes the schema, other compiled queries may have VDBE code that points to database objects that no longer exist or are in a different location in the database. Rather than running the risk of a bizarre runtime error later, SQLite invalidates all statements that have been compiled but not executed. They must be recompiled.

TRACING SQL

If you are having a hard time figuring out exactly what your program is doing with the database, you can track what SQL statements it has executed using `sqlite3_trace()`. Its declaration is as follows:

```
void *sqlite3_trace(sqlite3*, void(*xTrace)(void*,const char*), void*);
```

This function is analogous to putting a wiretap on a connection. You can use it to generate a log file of all SQL executed on a given connection as well provide helpful debugging information. Every SQL statement that is processed is passed to the callback function specified in the second argument. SQLite passes the data provided in the third argument of `sqlite3_trace()` to the first argument of the callback function.

Operational Control

The API provides several functions you can use to monitor and/or manage SQL commands at compile time and runtime. These functions allow you to install callback functions with which to monitor and control various database events as they happen.

Commit Hooks

The `sqlite3_commit_hook()` function allows you to monitor when transactions commit on a given connection. It is declared as follows:

```
void *sqlite3_commit_hook( sqlite3 *cnx,                /* database handle   */
                           int(*xCallback)(void *data), /* callback function */
                           void *data);                 /* application data  */
```

This function registers the callback function `xCallback`, which will be invoked whenever a transaction commits on the connection given by `cnx`. The third argument (`data`) is a pointer to application-specific data, which SQLite passes to the callback function. If the callback function returns a nonzero value, then the commit is converted into a rollback.

Passing a `NULL` value in for the callback function effectively disables the currently registered callback (if any). Also, only one callback can be registered at a time for a given connection. The return value for `sqlite3_commit_hook()` is `NULL` unless another callback function was previously registered, in which case the previous `data` value is returned.

Rollback Hooks

Rollback hooks are similar to commit hooks except that they watch for rollbacks for a given connection. Rollback hooks are registered with the following function:

```
void *sqlite3_rollback_hook(sqlite3 *cnx, void(*xCallback)(void *data), void *data);
```

This function registers the callback function `xCallback`, which will be invoked in the event of a rollback on `cnx`, whether by an explicit `ROLLBACK` command or by an implicit error or constraint violation. The callback is not invoked if a transaction is automatically rolled back because of the database connection being closed. The third argument (data) is a pointer to application-specific data, which SQLite passes to the callback function.

As in `sqlite3_commit_hook()`, each time you call `sqlite3_rollback_hook()`, the new callback function you provide will replace any currently registered callback function. If a callback function was previously registered, `sqlite3_rollback_hook()` returns the previous data argument.

Update Hooks

The `sqlite3_update_hook()` is used to monitor all `update`, `insert`, and `delete` operations on rows for a given database connection. It has the following form:

```
void *sqlite3_update_hook(
  sqlite3 *cnx,
  void(*)(void *, int, char const*, char const*, sqlite_int64),
  void *data);
```

The first argument of the callback function is a pointer to application-specific data, which you provide in the third argument. The callback function has the following form:

```
void callback ( void * data,
                int operation_code,
                char const *db_name,
                char const *table_name,
                sqlite_int64 rowid),
```

The operation_code argument corresponds to SQLITE_INSERT, SQLITE_UPDATE, and SQLITE_DELETE for insert, update, and delete operations, respectively. The third and fourth arguments correspond to the database name and table name the operation took place on. The final argument is the rowid of the affected row. The callback is not invoked for operations on system tables (for example, sqlite_master and sqlite_sequence). The return value is a pointer to the previously registered callback function's data argument, if it exists.

Authorizer Functions

Perhaps the most powerful event filter is sqlite3_set_authorizer(). It allows you to monitor and control queries as they are compiled. This function is declared as follows:

```
int sqlite3_set_authorizer(
  sqlite3*,
  int (*xAuth)( void*,int,
                const char*, const char*,
                const char*,const char*),
  void *pUserData
);
```

This routine registers a callback function that serves as an authorization function. SQLite will invoke the callback function at statement compile time (not at execution time) for various database events. The intent of the function is to allow applications to safely execute user-supplied SQL. It provides a way to restrict such SQL from certain operations (for example, anything that changes the database) or to deny access to specific tables or columns within the database.

For clarity, the form of the authorization callback function is as follows:

```
int auth( void*,         /* user data      */
          int,           /* event code     */
          const char*,   /* event specific */
          const char*,   /* event specific */
          const char*,   /* database name  */
          const char*    /* trigger or view name */ );
```

The first argument is a pointer to application-specific data, which is passed in on the fourth argument of sqlite3_set_authorizer(). The second argument to the authorization function will be one of the defined constants listed in Table 6-3. These values signify what kind of operation is to be authorized. The third and fourth arguments to the authorization function are specific to the event code. These arguments are listed with their respective event codes shown earlier in Table 6-2.

The fifth argument is the name of the database (main, temp, and so on) if applicable. The sixth argument is the name of the innermost trigger or view that is responsible for the access attempt or NULL if this access attempt is directly from top-level SQL.

The return value of the authorization function should be one of the constants SQLITE_OK, SQLITE_DENY, or SQLITE_IGNORE. The meaning of the first two values is consistent for all events—permit or deny the SQL statement. SQLITE_DENY will abort the entire SQL statement and generate an error.

The meaning of SQLITE_IGNORE is specific to the event in question. Statements that read or modify records generate SQLITE_READ or SQLITE_UPDATE events for each column the statement attempts to operate on. In this case, if the callback returns SQLITE_IGNORE, the column in question will be excluded from the operation. Specifically, attempts to read data from this column yield only NULL values, and attempts to update it will silently fail.

Download from Wow! eBook <www.wowebook.com>

Table 6-3. *SQLite Authorization Events*

Event Code	Argument 3	Argument 4
SQLITE_CREATE_INDEX	Index name	Table name
SQLITE_CREATE_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_INDEX	Index name	Table name
SQLITE_CREATE_TEMP_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_TRIGGER	Trigger name	Table name
SQLITE_CREATE_TEMP_VIEW	View name	NULL
SQLITE_CREATE_TRIGGER	Trigger name	Table name
SQLITE_CREATE_VIEW	View name	NULL
SQLITE_DELETE	Table name	NULL
SQLITE_DROP_INDEX	Index name	Table name
SQLITE_DROP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_INDEX	Index name	Table name
SQLITE_DROP_TEMP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_TRIGGER	Trigger name	Table name
SQLITE_DROP_TEMP_VIEW	View name	NULL
SQLITE_DROP_TRIGGER	Trigger name	Table name
SQLITE_DROP_VIEW	View name	NULL
SQLITE_INSERT	Table name	NULL
SQLITE_PRAGMA	Pragma name	First argument or NULL
SQLITE_READ	Table name	Column name
SQLITE_SELECT NULL		NULL

Continued

Event Code	Argument 3	Argument 4
SQLITE_TRANSACTION NULL		NULL
SQLITE_UPDATE	Table name	Column name
SQLITE_ATTACH	File name	NULL
SQLITE_DETACH	Database name	NULL

To illustrate this, the following example (the complete source of which is in `authorizer.c`) will create a table `foo`, defined as follows:

```
create table foo(x int, y int, z int)
```

It registers an authorizer function, which will do the following:

- Block reads of column `z`
- Block updates to column `x`
- Monitor `ATTACH` and `DETACH` database events
- Log database events as they happen

This is a rather long example, which uses the authorizer function to filter many different database events, so for clarity I am going to break the code into pieces. The authorizer function has the general form shown in Listing 6-7.

Listing 6-7. *Example Authorizer Function*

```
int auth( void* x, int type,
          const char* a, const char* b,
          const char* c, const char* d )
{
    const char* operation = a;

    printf( "    %s ", event_description(type));

    /* Filter for different database events
    ** from SQLITE_TRANSACTION to SQLITE_INSERT,
    ** UPDATE, DELETE, ATTACH, etc. and either allow or deny
    ** them.
    */

    return SQLITE_OK;
}
```

The first thing the authorizer looks for is a change in transaction state. If it finds a change, it prints a message:

```
if((a != NULL) && (type == SQLITE_TRANSACTION)) {
    printf(": %s Transaction", operation);
}
```

Next the authorizer filters events that result in a schema change:

```
switch(type) {
    case SQLITE_CREATE_INDEX:
    case SQLITE_CREATE_TABLE:
    case SQLITE_CREATE_TRIGGER:
    case SQLITE_CREATE_VIEW:
    case SQLITE_DROP_INDEX:
    case SQLITE_DROP_TABLE:
    case SQLITE_DROP_TRIGGER:
    case SQLITE_DROP_VIEW:
    {
        // Schema has been modified somehow.
        printf(": Schema modified");
    }
```

The next filter looks for read attempts (which are fired on a column-by-column basis). Here, all read attempts are allowed unless the column name is `z`, in which case the function returns `SQLITE_IGNORE`. This will cause SQLite to return `NULL` for any field in column `z`, effectively blocking access to its data.

```
if(type == SQLITE_READ) {
        printf(": Read of %s.%s ", a, b);

        /* Block attempts to read column z */
        if(strcmp(b,"z")==0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
    }
```

Next come `insert` and `update` filters. All `insert` statements are allowed. However, `update` statements that attempt to modify column `x` are denied. This will *not* block the `update` statement from executing; rather, it will simply filter out any attempt to update column `x`.

```
    if(type == SQLITE_INSERT) {
        printf(": Insert records into %s ", a);
    }

    if(type == SQLITE_UPDATE) {
        printf(": Update of %s.%s ", a, b);
```

```
        /* Block updates of column x */
        if(strcmp(b,"x")==0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
    }
```

Finally, the authorizer filters `delete`, `attach`, and `detach` statements and simply issues notifications when it encounters them:

```
    if(type == SQLITE_DELETE) {
        printf(": Delete from %s ", a);
    }

    if(type == SQLITE_ATTACH) {
        printf(": %s", a);
    }

    if(type == SQLITE_DETACH) {
        printf("-> %s", a);
    }

    printf("\n");
    return SQLITE_OK;
}
```

The (abbreviated) program is implemented as follows. As with the authorizer function, I will break it into pieces. The first part of the program connects to the database and registers the authorization function:

```
int main(int argc, char **argv)
{
    sqlite3 *db, *db2;
    char *zErr, *sql;
    int rc;

    /**  Setup */

    /* Connect to test.db */
    rc = sqlite3_open_v2("test.db", &db);

    /**  Authorize and test

    /* 1. Register the authorizer function */
    sqlite3_set_authorizer(db, auth, NULL);
```

Step 2 illustrates the transaction filter:

```
/* 2. Test transactions events */

printf("program : Starting transaction\n");
sqlite3_exec(db, "BEGIN", NULL, NULL, &zErr);

printf("program : Committing transaction\n");
sqlite3_exec(db, "COMMIT", NULL, NULL, &zErr);
```

Step 3 tests schema modifications by creating the test table `foo`:

```
/* 3. Test table events */

printf("program : Creating table\n");
sqlite3_exec(db, "create table foo(x int, y int, z int)", NULL, NULL, &zErr);
```

Step 4 tests read (`select`) and write (`insert`/`update`) control. It inserts a test record, selects it, updates it, and selects it again to observe the results of the `update`.

```
/* 4. Test read/write access */
printf("program : Inserting record\n");
sqlite3_exec(db, "insert into foo values (1,2,3)", NULL, NULL, &zErr);

printf("program : Selecting record (value for z should be NULL)\n");
print_sql_result(db, "select * from foo");

printf("program : Updating record (update of x should be denied)\n");
sqlite3_exec(db, "update foo set x=4, y=5, z=6", NULL, NULL, &zErr);

printf("program : Selecting record (notice x was not updated)\n");
print_sql_result (db, "select * from foo");

printf("program : Deleting record\n");
sqlite3_exec(db, "delete from foo", NULL, NULL, &zErr);

printf("program : Dropping table\n");
sqlite3_exec(db, "drop table foo", NULL, NULL, &zErr);
```

Several things are going on here. The program selects all records in the table, one of which is column `z`. We should see in the output that column `z`'s value is `NULL`. All other fields should contain data from the table. Next, the program attempts to update all fields, the most important of which is column `x`. The update should succeed, but the value in column `x` should be unchanged, because the authorizer denies it. This is confirmed on the following `select` statement, which shows that all columns were updated except for column `x`, which is unchanged. The program then drops the `foo` table, which should issue a schema change notification from the previous filter.

Step 5 tests the `attach` and `detach` database commands. The thing to notice here is how the authorizer function is provided with the name of the database and that you can distinguish the operations being performed under the attached database as opposed to the main database.

```
    /* 5. Test attach/detach */

    /* Connect to test2.db */
    rc = sqlite3_open_v2("test2.db", &db2);

    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db2));
        sqlite3_close(db2);
        exit(1);
     }

    sqlite3_exec(db2, "drop table foo2", NULL, NULL, &zErr);
    sqlite3_exec(db2, "create table foo2(x int, y int, z int)",
                      NULL, NULL, &zErr);

    printf("program : Attaching database test2.db\n");
    sqlite3_exec(db, "attach 'test2.db' as test2", NULL, NULL, &zErr);

    printf("program : Selecting record from attached database test2.db\n");
    sqlite3_exec(db, "select * from foo2", NULL, NULL, &zErr);

    printf("program : Detaching table\n");
    sqlite3_exec(db, "detach test2", NULL, NULL, &zErr);
```

Again for clarity, I will break up the program output into pieces. Upon executing it, the first thing we see is the transaction filter catching changes in transaction state:

```
program : Starting transaction
    SQLITE_TRANSACTION : BEGIN Transaction
program : Committing transaction
    SQLITE_TRANSACTION : COMMIT Transaction
program : Starting transaction
    SQLITE_TRANSACTION : BEGIN Transaction
program : Aborting transaction
    SQLITE_TRANSACTION : ROLLBACK Transaction
```

Next we see a schema change notification as the result of creating the test table `foo`. The interesting thing to notice here is all the other events that transpire in the `sqlite_master` table as a result of creating the table:

```
program : Creating table
    SQLITE_INSERT : Insert records into sqlite_master
    SQLITE_CREATE_TABLE : Schema modified
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
    SQLITE_UPDATE : Update of sqlite_master.type
    SQLITE_UPDATE : Update of sqlite_master.name
    SQLITE_UPDATE : Update of sqlite_master.tbl_name
```

```
    SQLITE_UPDATE : Update of sqlite_master.rootpage
    SQLITE_UPDATE : Update of sqlite_master.sql
    SQLITE_READ : Read of sqlite_master.ROWID
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
    SQLITE_READ : Read of sqlite_master.tbl_name
```

Next the program inserts a record, which the authorizer detects:

```
program : Inserting record
    SQLITE_INSERT : Insert records into foo
```

This is where things get more interesting. We are going to be able to see the authorizer block access to individual columns. The program selects all records from the `foo` table. We see the `SQLITE_SELECT` event take place, followed by the subsequent `SQLITE_READ` events generated for each attempted access of each column in the `select` statement. When it comes to column `z`, the authorizer denies access. Immediately following that, SQLite executes the statement, and `print_sql_result()` prints the column information and rows for the result set:

```
program : Selecting record (value for z should be NULL)
    SQLITE_SELECT
    SQLITE_READ : Read of foo.x
    SQLITE_READ : Read of foo.y
    SQLITE_READ : Read of foo.z -> DENIED
  Column: x (1/int)
  Column: y (1/int)
  Column: z (5/(null))
  Record: '1' '2' '(null)'
```

Look at what goes on with column `z`. Its value is `NULL`, which confirms that the authorizer blocked access. But also look at the column information. Although SQLite revealed the storage class of column `z`, it denied access to its declared type in the schema.

Next comes the update. Here we are interested in column `x`. The `update` statement will attempt to change every value in the record. But an update of column `x` will be denied:

```
program : Updating record (update of x should be denied)
    SQLITE_UPDATE : Update of foo.x -> DENIED
    SQLITE_UPDATE : Update of foo.y
    SQLITE_UPDATE : Update of foo.z
```

To confirm this, the program then selects the record to show what happened. The `update` did execute, but column `x` was not changed. Even more interestingly, while `z` was updated (trust me, it was), the authorizer will not let us see its value:

```
program : Selecting record (notice x was not updated)
    SQLITE_SELECT
    SQLITE_READ : Read of foo.x
    SQLITE_READ : Read of foo.y
    SQLITE_READ : Read of foo.z -> DENIED
  Column: x (1/int)
  Column: y (1/int)
  Column: z (5/(null))
  Record: '1' '5' '(null)'
```

Next the program deletes the record and drops the table. The latter operation generates all sorts of events on the `sqlite_master` table, just like when the table was created:

```
program : Deleting record
    SQLITE_DELETE : Delete from foo
program : Dropping table
    SQLITE_DELETE : Delete from sqlite_master
    SQLITE_DROP_TABLE : Schema modified
    SQLITE_DELETE : Delete from foo
    SQLITE_DELETE : Delete from sqlite_master
    SQLITE_READ : Read of sqlite_master.tbl_name
    SQLITE_READ : Read of sqlite_master.type
    SQLITE_UPDATE : Update of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.rootpage
```

Finally, the program creates another database on a separate connection and then attaches it on the main connection. The main connection then selects records from a table in the attached database, and we can see how the authorizer reports these operations as happening in the attached database:

```
program : Attaching database test2.db
    SQLITE_ATTACH : test2.db
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
program : Selecting record from attached database test2.db
    SQLITE_SELECT
    SQLITE_READ : Read of foo2.x
    SQLITE_READ : Read of foo2.y
    SQLITE_READ : Read of foo2.z -> DENIED
program : Detaching table
    SQLITE_DETACH -> test2
```

As you can see, `sqlite3_set_authorizer()` and its event-filtering capabilities are quite powerful. It gives you a great deal of control over what the user can and cannot do on a given database. You can monitor events and, if you want, stop them before they happen. Or, you can allow them to proceed but impose specific restrictions on them. `sqlite3_set_authorizer()` can be a helpful tool for dealing with

SQLITE_SCHEMA conditions. If you know you don't need any changes to your database schema in your application, you can simply install an authorizer function to deny any operations that attempt to modify the schema in certain ways.

■ **Caution** Keep in mind that denying schema changes in an authorizer is by no means a cure-all for SQLITE_SCHEMA events. You need to be careful about all the implications of denying various changes to the schema. Just blindly blocking all events resulting in a schema change can restrict other seemingly legitimate operations such as VACUUM.

HELP FOR INTERACTIVE PROGRAMS

SQLite provides two functions that make it easier to work with interactive programs. The first is sqlite3_interrupt(), which is declared as follows:

```
void sqlite3_interrupt(sqlite3* /* connection handle */);
```

sqlite3_interrupt() causes any pending database operation on the given connection to abort and return at its earliest opportunity. This routine is design to be called in response to a user interrupt action such as clicking a Cancel button in a graphical application or pressing Ctrl+C in a command-line program. It is intended to make it easier to accommodate cases where the user wants a long query operation to halt immediately. For a command-line program, you might put this function in a signal handler or an event handler in a graphical application.

Another function that can be useful for interactive programs is sqlite3_progress_handler(). It is somewhat related in functionality to the sqlite3_interrupt() function, but with a few twists. It is declared as follows:

```
void sqlite3_progress_handler( sqlite3*,      /* connection handle     */
                               int frq,       /* frequency of callback */
                               int(*)(void*), /* callback function     */
                               void*);        /* application data      */
```

This function installs a callback function that will be invoked during calls to sqlite3_exec(), sqlite3_step(), and sqlite3_get_table(). The intent of this function is to give applications a way to provide feedback to the user during long-running queries.

The second argument (frq) specifies the frequency of the callback in terms of VDBE instructions. That is, the progress callback (provided in the third argument) will be called for every frq VDBE instruction performed in query execution. If a call to sqlite3_exec(), sqlite3_step(), or sqlite3_get_table() requires less than frq instructions to be executed, then the progress callback will not be invoked. The fourth argument is a pointer to application-specific data, which is passed back to the progress handler (as its one and only argument). If NULL is provided for the callback function argument, then the currently installed progress handler (if any) is disabled.

If the progress callback returns a nonzero value, the current query will be immediately terminated, and any database changes will be rolled back. If the query was part of a larger transaction, then the transaction is not rolled back and remains active. The `sqlite3_exec()` call in that case will return `SQLITE_ABORT`.

Threads

SQLite has enjoyed threading support for many releases. Although normal defensive coding techniques in multithreaded code apply to SQLite as well, there are a few additional precautions to take.

One limitation that is somewhat thread-related pertains to the `fork()` system call on Unix. You should never try to pass a connection across a `fork()` call to a child process. It will not work correctly.

Shared Cache Mode

Starting with SQLite version 3.3.0, SQLite supports something called *shared cache mode*, which allows multiple connections in a single process to use a common page cache, as shown in Figure 6-1. This feature is designed for embedded servers where a single thread can efficiently manage multiple database connections on behalf of other threads. The connections in this case share a single page cache as well as a different concurrency model. Because of the shared cache, the server thread can operate with significantly lower memory usage and better concurrency than if each thread were to manage its own connection. Normally, each connection allocates enough memory to hold 2,000 pages in the page cache. Rather than each thread consuming that much memory with its own connection, it shares that memory with other threads using a single page cache.

In this model, threads rely on a server thread to manage their database connections for them. A thread sends SQL statements to the server through some communication mechanism; the server executes them using the thread's assigned connection and then sends the results back. The thread can still issue commands and control its own transactions; only its actual connection exists in, and is managed by, another thread.

Download from Wow! eBook <www.wowebook.com>

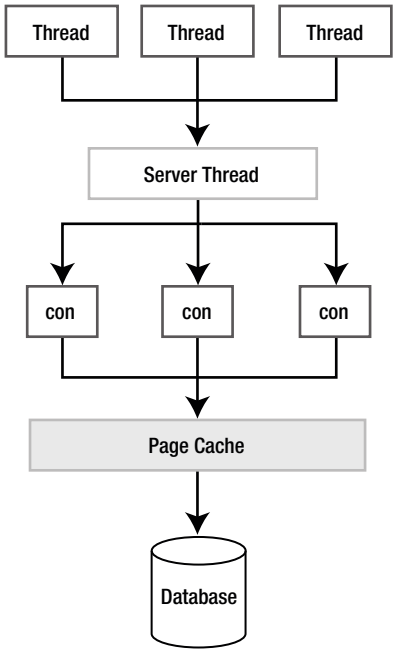


Figure 6-1. *The shared cache model*

Connections in shared cache mode use a different concurrency model and isolation level. Furthermore, an option exists to allow connections in a shared cache can see what their kindred connections have changed. If the pragma `read_committed` is set to `true`, this kind of change inspection is possible. That is, there can be both multiple readers and an active writer in the database at the same time. That writer can write to the database—performing complete write transactions—without having to wait for other readers to clear. Therefore, data can change in the database within a reader's transaction. Although SQLite normally runs in a serialized isolation level, meaning that readers and writers always see a consistent view of the database, in shared cache mode, readers are subject to the database changing underneath them.

There are some control measures in place to keep connections out of one another's way. By default, shared cache mode uses *table locks* to keep reader connections and writer connections separated. Table locks are not the same as database locks and exist only within connections of the shared cache. Whenever a connection reads a table, SQLite first tries to get a read lock on it. The same is true for writing to a table. A connection may not write to a table that another connection has a read lock on, and vice versa. Furthermore, table locks are tied to their respective connection and last for the duration of its transaction.

Read Uncommitted Isolation Level

Although table locks help keep connections in shared cache mode out of one another's way, it is possible for connections to elect not to be in the way at all. That is, a connection can choose to have a *read-uncommitted* isolation level by using the `read_uncommitted` pragma. If it is set to `true`, then the

connection will not put read locks on the tables it reads. Therefore, another writer can actually change a table as the connection reads it. This can lead to inconsistent query results, but it also means that a connection in read-uncommitted mode can neither block nor be blocked by any other connections.

Some large changes have happened as of version 3.7 of SQLite, with the introduction of Write-Ahead Log mode. We'll cover Write-Ahead Logging in detail in Chapter 11.

SCHEMA CHANGES IN SHARED CACHE MODE

There are additional things to consider in shared cache mode when a thread wants to modify the database schema. As it turns out, before SQLite issues a table lock, it first gets a read table lock on the `sqlite_master` table. For a thread to alter the schema, it must first get a write table lock on sqlite_master. It can do this only if there are no read locks on it from other tables. When a thread does get a write lock on `sqlite_master`, then no other threads are allowed to compile SQL statements during that time. They will get `SQLITE_SCHEMA` errors.

Unlock Notification

If the thought of operating in the wilds of read-uncommitted isolation doesn't appeal to you, SQLite provides for an alternative where shared cache mode is in use. Recent versions of SQLite have included the `sqlite3_unlock_notify()` function to provide additional flexibility in locking situations. The function profile looks as follows:

```
int sqlite3_unlock_notify(
  sqlite3 *pBlocked,                          /* Waiting connection */
  void (*xNotify)(void **apArg, int nArg),    /* Callback function to invoke */
  void *pNotifyArg                            /* Argument to pass to xNotify */
)
```

If your code encounters a failure to gain a shared lock because of a contending lock, `SQLITE_LOCKED` will be returned. At this point, you can call `sqlite3_unlock_notify()` to allow registering of your callback function `xNotify` (the second parameter) with any desired parameters to that function (access via the `pNotifyArg` pointer as the third parameter) against your blocked connection (the first parameter).

The connection that holds the blocking lock will trigger the `xNotify` callback as part of the `sqlite3_step()` or `sqlite3_close()` call that completes its transaction. It is also possible, particularly within a multithreaded application, that in the time it takes you to invoke `sqlite3_unlock_notify()`, the competing transaction will have completed. In this case, your callback function will be initiated immediately from within `sqlite3_unlock_notify()`.

There are some caveats to the use of `sqlite3_unlock_notify()`, including the following:

- You can register only one unlock/notify callback per blocked connection (newer calls replace any existing callback).
- `sqlite3_unlock_notify()` is not reentrant, meaning you should *not* include other SQLite function calls in your callback function. You're likely to crash your application.

- Special care must be taken if you plan to use with `drop table` or `drop index` commands. Because these can be "blocked" by `select` statements, there is no real blocking transaction to include as the `pBlocked` parameter. Refer to the details on the SQLite web site for using the extended return codes to detect and handle this scenario at `www.sqlite.org/c3ref/unlock_notify.html`.

Threads and Memory Management

Since shared cache mode is about conserving memory, SQLite has several functions associated with threads and memory management. They allow you to specify an advisory heap limit—a soft heap limit—as well as manually initiate memory cleanups. These functions are as follows:

```
void sqlite3_soft_heap_limit(int N);
int sqlite3_release_memory(int N);
```

The `sqlite3_soft_heap_limit()` function sets the current soft heap limit of the calling thread to N bytes. If the thread's heap usage exceeds N, then SQLite automatically calls `sqlite3_release_memory()`, which attempts to free at least N bytes of memory from the caches of all database connections associated with the calling thread. The return value of `sqlite3_release_memory()` is the number of bytes actually freed.

These routines are no-ops unless you have enabled memory management by compiling SQLite with the `SQLITE_ENABLE_MEMORY_MANAGEMENT` preprocessor directive.

Summary

The core C API contains everything you need to process SQL commands and then some. It contains a variety of convenient query methods that are useful in different ways. These include `sqlite3_exec()` and `sqlite3_get_table()`, which allow you to execute commands in a single function call. The API includes many utility functions as well, allowing you to determine the number of affected records in a query, get the last inserted `ROWID` of an `INSERT` statement, trace SQL commands run on a connection, and conveniently format strings for SQL statements.

The `sqlite3_prepare_v2()`, `sqlite3_step()`, and `sqlite3_finalize()` methods provide you with a lot of flexibility and control over statement execution through the use of statement handles. Statement handles provide more detailed row and column information, the convenience of bound parameters, and the ability to reuse prepared statements, avoiding the overhead of query compilation.

SQLite provides a variety of ways to manage runtime events through the use of event filters. Its commit, rollback, and update hooks allow you to monitor and control specific changes in database state. You can watch changes to rows and columns as they happen and use an authorizer function to monitor and restrict what queries can do as they are compiled.

And believe it or not, you've seen only half of the API! Actually, you've seen more like three-quarters, but you are likely to find that what is in the next chapter—user-defined functions, aggregates, and collations—is every bit as useful and interesting as what is in this one.

CHAPTER 7

The Extension C API

This chapter is about teaching SQLite new tricks. The previous chapter dealt with generic database work; this chapter is about being creative. The Extension API offers several basic ways to extend or customize SQLite, through the creation of user-defined *functions*, *aggregates*, *collation sequences*, and virtual tables. At a lower level, many more options present themselves, including interchangeable VFSs, start-time changeable page cache, and malloc and mutex implementation. We'll cover the first three options in this chapter: functions, aggregates, and collation sequences.

User-defined functions are SQL functions that map to some implementation that you write. They are callable from within SQL. For example, you could create a function `hello_newman()` that returns the string `'Hello Jerry'` and, once it is registered, call it from SQL as follows:

```
sqlite > select hello_newman() as reply;
reply
------------------
'Hello Jerry'
```

This is a special version of the SQLite command-line program we've customized that includes `hello_newman()`.

Aggregates are a special form of function and work in much the same way except that they operate on sets of records and return aggregated values or expressions computed over a particular column in the set. Or they may be computed from multiple columns. Standard aggregates that are built into SQLite include `SUM()`, `COUNT()`, and `AVG()`, for example.

Collations are methods of comparing and sorting text. That is, given two strings, a collation would decide which one is greater or whether the two are equal. The default collation in SQLite is `BINARY`—it compares strings byte by byte using `memcmp()`. Although this collation happens to work well for words in the English language (and other languages that use UTF-8 encoding), it doesn't necessarily work so well for other languages that use different encodings. Thus, SQLite includes the ability to build your own user-defined collations to better handle those alternative languages.

This chapter covers each of these three user-defined extension facilities and their associated API functions. As you will see, the user-defined extensions API can prove to be a powerful tool for creating nontrivial features that extend SQLite.

All of the code in this chapter is included in examples that you can obtain online from the Apress web site (`www.apress.com`). The examples for this particular chapter are located in the `ch7` subdirectory of the `examples` folder. We will specifically point out all the source files corresponding to the examples as they are introduced in the text. Some of the error-checking code in the figures has been taken out for the sake of clarity.

Also, many examples in this chapter use simple convenience functions from a common library written to help simplify the examples. We will point out these functions and explain what they do as they are introduced in the text. The common code is located in the `common` subdirectory of the `examples` folder and must first be built before the examples are built. Please refer to the `README` file in the `examples` folder for details on building the common library.

Finally, to clear up any confusion, the terms *storage class* and *data type* are interchangeable. You know that SQLite stores data internally using five different data types, or storage classes. These classes are `INTEGER`, `FLOAT`, `TEXT`, `BLOB`, and `NULL`, as described in Chapter 4. Throughout the chapter, we'll use the term that best fits the context. You'll generally see the term *data type*, because it's most broadly applicable and easier for readers from different backgrounds to follow. When speaking about how SQLite handles a specific value internally, the term *storage class* seems to work better. In either case, you can equate either term as describing a specific form or representation of data.

The API

The basic method for implementing functions, aggregates, and collations consists of implementing a callback function and then registering it in your program. Once registered, your functions can be used from SQL. Functions and aggregates use the same registration function and similar callback functions. Collation sequences, while using a separate registration function, are almost identical and can be thought of as a particular class of user-defined function.

The lifetime of user-defined aggregates, functions, and collations is transient. They are registered on a connection-by-connection basis; they are *not* stored in the database. It is up to you to ensure that your application loads your custom extensions and registers them on each connection. Sometimes you may find yourself thinking of your extensions in terms of stored procedures and completely forget about the fact that they don't actually reside in the database. Extensions exist in libraries, not in databases, and are restricted to the life of your program. If you don't grasp this concept, you may do something like try to create a trigger (which is stored in the database) that references your extensions (which aren't stored in the database) and then turn around and try to call it from the SQLite shell or from an application that knows nothing of your extensions. In this case, you will find that either it triggers an error or nothing happens. Extensions require your program to register them on *each* and *every* connection; otherwise, that connection's SQL syntax will know nothing about them.

Registering Functions

You can register functions and aggregates on a connection using `sqlite3_create_function()`, which is declared as follows:

```
int sqlite3_create_function(
  sqlite3 *cnx,               /* connection handle                     */
  const char *zFunctionName,  /* function/aggregate name in SQL        */
  int nArg,                   /* number of arguments. -1 = unlimited. */
  int eTextRep,               /* encoding (UTF8, 16, etc.)             */
  void *pUserData,            /* application data, passed to callback */
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

```
int sqlite3_create_function16(
  sqlite3 *cnx,
  const void *zFunctionName,
  int nArg,
  int eTextRep,
  void *pUserData,
  void (*xFunc)(sqlite3_context*, int args, sqlite3_value**),
  void (*xStep)(sqlite3_context*, int args, sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

As with our examples in the previous chapter, we've included both the UTF-8 and UTF-16 versions of this function to underscore that the SQLite API has functions that support both encodings. From now on, we will only refer to the UTF-8 versions for the sake of brevity.

The arguments for `sqlite3_create_function()` are defined as follows:

- `cnx`: The database connection handle. Functions and aggregates are connection specific. They must be registered on the connection as it is opened in order to be used.
- `zFunctionName`: The function name as it will be called in SQL. Up to 255 bytes in length.
- `nArg`: The number of arguments expected by your function. SQLite will enforce this as the exact number of arguments that must be provided to the function. If you specify -1, SQLite will allow a variable number of arguments.
- eTextRep: The preferred text encoding. Allowable values are `SQLITE_UTF8`, `SQLITE_UTF16`, `SQLITE_UTF16BE`, `SQLITE_UTF16LE`, and `SQLITE_ANY`. This field is a hint to SQLite used only to help it choose between multiple implementations of the same function. `SQLITE_ANY` means "no preference." In other words, the implementation works equally as well with any text encoding. When in doubt, use `SQLITE_ANY` here.
- `pUserData`: Application-specific data. This data is made available through the callback functions specified in `xFunc`, `xStep`, and `xFinal` below. Unlike other functions, which receive the data as a void pointer in the argument list, these functions must use a special API function to obtain this data.
- `xFunc`: The function callback. This is the actual implementation of the SQL function. Functions only provide this callback and leave the `xStep` and `xFinal` function pointers `NULL`. These latter two callbacks are exclusively for aggregate implementations.
- `xStep`: The aggregate step function. Each time SQLite processes a row in an aggregated result set, it calls `xStep` to allow the aggregate to process the relevant field value(s) of that row and include the result in its aggregate computation.
- `xFinal`: The aggregate finalize function. When all rows have been processed, SQLite calls this function to allow the aggregate to conclude its processing, which usually consists of computing the final value and optionally cleaning up.

You can register multiple versions of the same function differing only in the encoding (`eTextRep` argument) and/or the number of arguments (`nArg` argument), and SQLite will automatically select the best version of the function for the case in hand.

The Step Function

The function (callback) and aggregate step functions are identical and are declared as follows:

```
void fn(sqlite3_context* ctx, int nargs, sqlite3_value** values)
```

The `ctx` argument is the function/aggregate context. It holds state for a particular function call instance and is the means through which you obtain the application data (`pUserData`) argument provided in `sqlite3_create_function()`. The user data is obtained from the context using `sqlite3_user_data()`, which is declared as follows:

```
void *sqlite3_user_data(sqlite3_context*);
```

For functions, this data is shared among all calls to like functions, so it is not really unique to a particular instance of function call. That is, the same `pUserData` is passed or shared among all instances of a given function. Aggregates, however, can allocate their own state for each particular instance using `sqlite3_aggregate_context()`, declared as follows:

```
void *sqlite3_aggregate_context(sqlite3_context*, int nBytes);
```

The first time this routine is called for a particular aggregate, a chunk of memory of size `nBytes` is allocated, zeroed, and associated with that context. On subsequent calls with the same context (for the same aggregate instance), this allocated memory is returned. Using this, aggregates have a way to store state in between calls in order to accumulate data (which, when you think about it, is the general purpose for aggregating something). When the aggregate completes the `final()` callback, the memory is automatically freed by SQLite.

■ **Note** One of the things you will see throughout the API is the use of user data in void pointers. Since many parts of the API involve callback functions, these simply serve as a convenient way to maintain state when implementing said callback functions.

The `nargs` argument of the callback function contains the number of arguments passed to the function.

Return Values

The `values` argument is an array of SQLite value structures that are handles to the actual argument values. The actual data for these values is obtained using the family of `sqlite3_value_`*`xxx`*`()` functions, which have the following form:

```
xxx sqlite3_value_xxx(sqlite3_value* value);
```

where *xxx* is the C data type to be returned from the `value` argument. If you read Chapter 6, you may be thinking that these functions have a striking resemblance to the `sqlite3_column_xxx()` family of functions—and you'd be right. They work in the same way, even down to the difference between the way scalar and array values are obtained. The functions for obtaining scalar values are as follows:

```
int sqlite3_value_int(sqlite3_value*);
sqlite3_int64 sqlite3_value_int64(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
```

The functions used to obtain array values are as follows:

```
int sqlite3_value_bytes(sqlite3_value*);
const void *sqlite3_value_blob(sqlite3_value*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
```

The `sqlite3_value_bytes()` function returns the amount of data in the value buffer for a `BLOB`. The `sqlite3_value_blob()` function returns a pointer to that data. Using the size and the pointer, you can then copy out the data. For example, if the first argument in your function was a `BLOB` and you wanted to make a copy, you would do something like this:

```
int len;
void* data;

len = sqlite3_value_bytes(values[0]);
data = sqlite3_malloc(len);
memcpy(data, sqlite3_value_blob(values[0]), len);
```

Just as the `sqlite3_column_xxx()` functions have `sqlite_column_type()` to provide the column types, the `sqlite3_value_xxx()` functions likewise have `sqlite3_value_type()`, which works in the same way. It is declared as follows:

```
int sqlite3_value_type(sqlite3_value*);
```

This function returns one of the following values, which correspond to SQLite's internal, storage classes (data types), defined as follows:

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_TEXT     3
#define SQLITE_BLOB     4
#define SQLITE_NULL     5
```

This covers the basic workings of the function/aggregate interface. We can now dive into some examples and practical applications. The Collation interface is so similar that it will be immediately understandable to you now that you've reviewed the interface for functions and aggregates.

Functions

Let's start our custom function development with a trivial example, one that is very easy to follow. Let's implement `hello_newman()`, but with a slight twist. We want to include using function arguments in the example, so `hello_newman()` will take one argument: the name of the person addressing him. It will work as follows:

```
sqlite > select hello_newman('Jerry') as reply;
reply
------------------
Hello Jerry

sqlite > select hello_newman('Kramer') as reply;
reply
------------------
Hello Kramer

sqlite > select hello_newman('George') as reply;
reply
------------------
Hello George
```

Listing 7-1 shows the basic program (taken from `hello_newman.c`). Some of the error-checking code has been removed for clarity.

Listing 7-1. *The `hello_newman()` Test Program*

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db;

    sqlite3_open_v2("test.db", &db);
    sqlite3_create_function( db, "hello_newman", 1, SQLITE_UTF8, NULL,
                             hello_newman, NULL, NULL);

    /* Log SQL as it is executed. */
    log_sql(db,1);

    /* Call function with one text argument. */
    fprintf(stdout, "Calling with one argument.\n");
    print_sql_result(db, "select hello_newman('Jerry')");

    /* Call function with two arguments.
    ** It will fail as we registered the function as taking only one argument.*/
    fprintf(stdout, "\nCalling with two arguments.\n");
    print_sql_result(db, "select hello_newman ('Jerry', 'Elaine')");
```

Download from Wow! eBook <www.wowebook.com>

```
    /* Call function with no arguments. This will fail too */
    fprintf(stdout, "\nCalling with no arguments.\n");
    print_sql_result(db, "select hello_newman()");

    /* Done */
    sqlite3_close(db);

    return 0;
}
```

This program connects to the database, registers the function, and then calls it three times. The callback function is implemented as follows:

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    const char *msg;

    /* Generate Newman's reply */
    msg = sqlite3_mprintf("Hello %s", sqlite3_value_text(values[0]));

    /* Set the return value. Have sqlite clean up msg w/ sqlite_free(). */
    sqlite3_result_text(ctx, msg, strlen(msg), sqlite3_free);
}
```

Running the program yields the following output:

```
Calling with one argument.
  TRACE: select hello_newman('Jerry')
hello_newman('Jerry')
---------------------
Hello Jerry

Calling with two arguments.
execute() Error: wrong number of arguments to function hello_newman()

Calling with no arguments.
execute() Error: wrong number of arguments to function hello_newman()
```

The first call consists of just one argument. Since the example registered the function as taking exactly one argument, it succeeds. The second call attempts to use two arguments, which fails. The third call uses no arguments and also fails.

And there you have it: a SQLite extension function. This does bring up a few new functions that we have not addressed yet. We've created several helper functions for use with the book and to aid you in your SQLite work. You'll find these included for download on the Apress web site. Many of these functions' definitions can be found in the source file `util.c`. The function `log_sql()` simply calls `sqlite3_trace()`, passing in a tracing function that prefixes the traced SQL with the word *TRACE*. We use this so you can see what is happening in the example as the SQL is executed. The second function is `print_sql_result()`, which has the following declaration:

```
int print_sql_result(sqlite3 *db, const char* sql, ...)
```

This is a simple wrapper around `sqlite3_prepare_v2()` and friends that executes a SQL statement and prints the results.

Return Values

Our `hello_newman()` example introduces a new SQLite function: `sqlite3_result_text()`.This function is just one of a family of `sqlite_result_xxx()` functions used to set return values for user-defined functions and aggregates. The scalar functions are as follows:

```
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
```

These functions simply take a (second) argument of the type specified in the function name and set it as the return value of the function. The array functions are as follows:

```
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
```

These functions take array data and set that data as the return value for the function. They have the following general form:

```
void sqlite3_result_xxx(
    sqlite3_context *ctx,   /* function context */
    const xxx* value,       /* array value */
    int len,                /* array length */
    void(*free)(void*));    /* array cleanup function */
```

Here *xxx* is the particular array type—`void` for `BLOB`s or `char` for `TEXT`. Again, if you read Chapter 6, you may find these functions suspiciously similar to the `sqlite3_bind_xxx()` functions, which they are. Setting a return value for a function is, for all intents and purposes, identical to binding a value to a parameterized SQL statement. You could even perhaps refer to it as "binding a return value."

Arrays and Cleanup Handlers

Just as in binding array values to SQL statements, these functions work in the same way as `sqlite3_bind_xxx()`. They require a pointer to the array, the length (or size) of the array, and a function pointer to a cleanup function. This cleanup function pointer can be assigned the same predefined meanings as in `sqlite3_bind_xxx()`:

```
  #define SQLITE_STATIC      ((void(*)(void *))0)
  #define SQLITE_TRANSIENT   ((void(*)(void *))-1)
```

`SQLITE_STATIC` means that the array memory resides in unmanaged space, and therefore SQLite does not need to make a copy of the data for use, nor should it attempt to clean it up. `SQLITE_TRANSIENT` tells SQLite that the array data is subject to change, and therefore SQLite needs to make its own copy

using `sqlite3_malloc()`. The allocated memory will be freed when the function returns. The third option is to pass in an actual function pointer to a cleanup function of the following form:

```
void cleanup(void*);
```

In this case, SQLite will call the cleanup function after the user-defined function completes. This is the method used in the previous example. We used `sqlite3_mprintf()` to generate Newman's reply, which allocated a string on the heap. We then passed in `sqlite3_free()` as the cleanup function to `sqlite3_result_text()`, which could free the string memory when the extension function completed.

Error Conditions

Sometimes functions encounter errors, in which case the return value should be set appropriately. This is what `sqlite3_result_error()` is for. It is declared as follows:

```
void sqlite3_result_error(
    sqlite3_context *ctx, /* the function context */
    const char *msg,      /* the error message */
    int len);             /* length of the error message */
```

This tells SQLite that there was an error, the details of which are contained in `msg`. SQLite will abort the command that called the function and set the error message to the value contained in `msg`.

Returning Input Values

Sometimes, you may want to pass back an argument as the return value in the same form. Rather than you having to determine the argument's type, extract its value with the corresponding s`qlite3_column_`*`xxx`*`()` function, and then turn right around and set the return value with the appropriate `sqlite3_result_`*`xxx`*`()` function, the API offers `sqlite3_result_value()` so you can do all of this in one fell swoop. It is declared as follows:

```
void sqlite3_result_value(
    sqlite3_context *ctx,   /* the function context */
    sqlite3_value* value);   /* the argument value   */
```

For example, say you wanted to create a function `echo()` that spits back its first argument. The implementation is as follows:

```
void echo(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    sqlite3_result_value(ctx, values[0]);
}
```

It would work from SQL as follows:

```
sqlite > select echo('Hello Jerry') as reply;
reply
------------------
Hello Jerry
```

Even better, `echo()` works equally well with arguments of any storage class and returns them accordingly:

```
sqlite> select echo(3.14) as reply, typeof(echo(3.14)) as type;
reply       type
----------  ----------
3.14        real
sqlite> select echo(X'0128') as reply, typeof(echo(X'0128')) as type;
reply       type
----------  ----------
(?         blob
sqlite> select echo(NULL) as reply, typeof(echo(NULL)) as type;
reply       type
----------  ----------
            null
```

Aggregates

Aggregates are only slightly more involved than functions. Whereas you had to implement only one callback function to do the work in user-defined functions, you have to implement both a step function to compute the ongoing value as well as a finalizing function to finish everything off and clean up. Figure 7-1 shows the general process. It still isn't hard, though. Like functions, one good example will be all you need to get the gist of it.

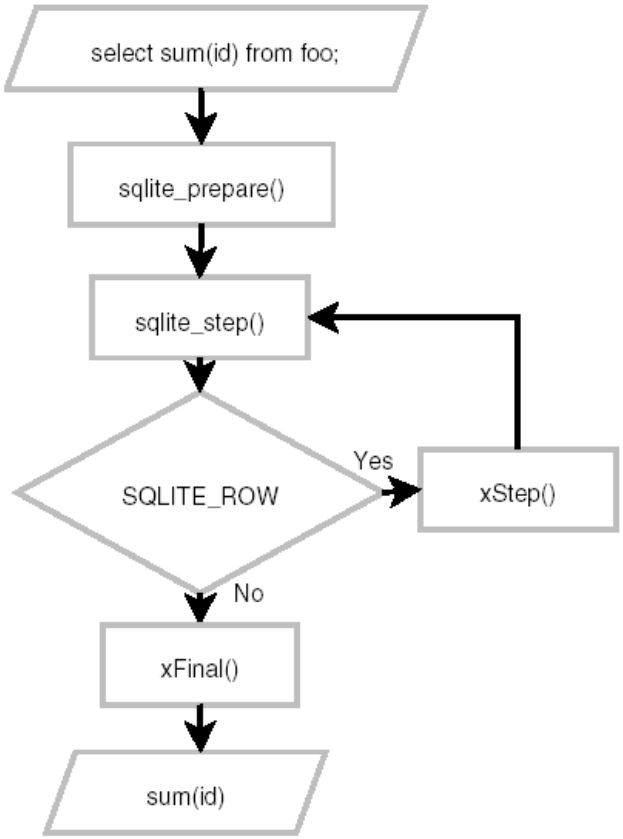


Figure 7-1. *Query processing with aggregates*

Registration Function

As mentioned before, aggregates and functions use the same registration function, `sqlite3_create_function()`, which is listed here again for convenience:

```
int sqlite3_create_function(
  sqlite3 cnx*,               /* connection handle                     */
  const char *zFunctionName,  /* function/aggregate name in SQL        */
  int nArg,                   /* number of arguments. -1 = unlimited. */
  int eTextRep,               /* encoding (UTF8, 16, etc.)             */
  void *pUserData,            /* application data, passed to callback */
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

You do exactly the same thing here as you do with functions, but rather than proving a callback for `xFunc`, you leave it `NULL` and provide functions for `xStep` and `xFinal`.

A Practical Example

The best way to illustrate implementing aggregates is by example. Here we will implement the aggregate `str_agg()`. The idea of `str_agg()` is to act much like the `strcat()` C function, where two strings are concatenated together (the eagle-eyed will spot that newer versions of SQLite include a function with a similar purpose, `group_concat()`). But for our purposes, we want to aggregate any number of strings involved in grouping and aggregations at the behest of your SQL in SQLite. You might well wonder whether you really need such a string contatenation aggregation feature in SQLite, given you could write your own in C. One advantage of the `str_agg()` approach is to enable semantically clear and easily understood SQL to be written, rather than having to shuffle back and forth between SQL and C to achieve the string aggregation. The other benefit of this example is, of course, to show you *how* to aggregate. Listing 7-2 shows our example `str_agg()` function (taken from `str_agg.c`).

Listing 7-2. *The* `str_agg()` *Test Program*

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db; char *sql;

    sqlite3_open_v2("foods.db", &db);

    /* Register aggregate. */
    fprintf(stdout, "Registering aggregate str_agg()\n");

    /* Turn SQL tracing on. */
    log_sql(db, 1);

    /* Register aggregate. */
    sqlite3_create_function( db, "str_agg", 1, SQLITE_UTF8, db,
                             NULL, str_agg_step, str_agg_finalize);

    /* Test. */
    fprintf(stdout, "\nRunning query: \n");
    sql = "select season, str_agg(name, ', ') from episodes group by season";
    print_sql_result(db, sql);

    sqlite3_close(db);

    return 0;
}
```

Note that our step function is called `str_agg_step()`, and our finalizing function is called `str_agg_finalize()`. Also, this aggregate is registered as taking only one argument.

The Step Function

Listing 7-3 shows the `str_agg_step()` function.

Listing 7-3. *The `str_agg()` Step Function*

```
void str_agg_step(sqlite3_context* ctx, int ncols, sqlite3_value** values)
{
  SAggCtx *p = (SAggCtx *) sqlite3_aggregate_context(ctx, sizeof(*p));

  static const char delim [] = ", ";

  char *txt = sqlite3_value_text(values[0]);

  int len = strlen(txt);

  if (!p->result) {
    p->result = sqlite_malloc(len + 1);
    memcpy(p->result, txt, len + 1);
    p->chrCnt = len;
  }
  else
  {
    const int delimLen = sizeof(delim);
    p->result = sqlite_realloc(p->result, p->chrCnt + len + delimLen + 1);
    memcpy(p->result +  p->chrCnt, delim, delimLen);
    p->chrCnt += delimLen;
    memcpy(p->result +  p->chrCnt, txt, len + 1);
    p->chrCnt += len;
  }
}
```

The value `SAggCtx` is a struct that is specific to this example and is defined as follows:

```
typedef struct SAggCtx SAggCtx;
struct SAggCtx {
  int chrCnt;
  char *result;
};
```

Our structure serves as our state between aggregate iterations (calls to the step function), maintaining the growing concatenation of text, and the character count.

The Aggregate Context

The first order of business is to retrieve the structure for the given aggregate instance. This is done using `sqlite3_aggregate_context()`. As mentioned earlier, the first call to this function allocates the data for the given context and subsequent calls retrieve it. The structure memory is automatically freed when the aggregate completes (after `str_agg_finalize()` is called). Note that `sqlite3_aggregate_context()` might return `NULL` if the system is low on memory. Such error checking is omitted in this example, for brevity,

but a real implementation should always check the result of `sqlite3_aggregate_context()` and invoke `sqlite3_result_error_nomem()` if the result is `NULL`. In the `str_agg_step()` function, the text value to be aggregated is retrieved from the first argument using `sqlite3_value_text()`. This value is then added to the `result` member of the `SAggCtx` struct, which stores the intermediate concatenation. Note again that `sqlite3_value_text()` might return a `NULL` pointer if the argument to the SQL function is `NULL` or if the system is low on memory. A real implementation should check for this case in order to avoid a `segfault`.

The Finalize Function

Each record in the materialized result set triggers a call to `str_agg_step()`. After the last record is processed, SQLite calls `str_agg_finalize()`, which is implemented as shown in Listing 7-4.

Listing 7-4. *The `sum_agg()` Finalize Function*

```
void str_agg_finalize(sqlite3_context* ctx)
{
  SAggCtx *p = (SAggCtx *) sqlite3_aggregate_context(ctx, sizeof(*p));
  if( p && p->result ) sqlite3_result_text(ctx, p->result, p->chrCnt, sqlite_free);
}
```

The `str_agg_finalize()` function basically works just like a user-defined function callback. It retrieves the `SAggCtx` struct and, if it is not `NULL`, sets the aggregate's return value to the value stored in the struct's `result` member using `sqlite3_result_text()`. The `sqlite3_aggregate_context()` routine might return `NULL` if the system is low on memory, or the `p->zResult` field might be `NULL` if the step function was never called. In the former case, SQLite is going to return the `SQLITE_NOMEM` error so it does not really matter whether `sqlite3_result_text()` is ever called. In the latter case, since no `sqlite3_result_xxx()` function is ever invoked for the aggregate, the result of the aggregate will be the SQL `NULL`.

Results

The program produces the following output (abbreviated here to conserve paper):

```
Registering aggregate str_agg()

Running query:
  TRACE: select season, str_agg(name, ', ') from episodes group by season
…

season str_agg(name, ', ')
------ ------------------------------------------------------------------
0      Good News Bad News
1      Male Unbonding, The Stake Out, The Robbery, The Stock Tip
2      The Ex-Girlfriend, The Pony Remark, The Busboy, The Baby Shower, …
3      The Note, The Truth, The Dog, The Library, The Pen, The Parking Garage …
…
```

As you can see, the `select` statement pulls back the data from the `episode` table, and the `str_agg()` aggregate concatenates the episode names per season. You can think of this as something similar to cross-tabs or pivots in spreadsheets and other tools.

And that is the concept of aggregates in a nutshell. They are so similar to functions that there isn't much to talk about once you are familiar with functions.

Collations

As stated before, the purpose of collations is to sort strings. But before starting down that path, it is important to recall that SQLite's manifest typing scheme allows varying data types to coexist in the same column. For example, consider the following SQL (located in `collate1.sql`):

```
.headers on
.m column
create table foo(x);
insert into foo values (1);
insert into foo values (2.71828);
insert into foo values ('three');
insert into foo values (X'0004');
insert into foo values (null);

select quote(x), typeof(x) from foo;
```

Feeding it to the SQLite CLP will produce the following:

```
C:\temp\examples\ch7> sqlite3 < collate.sql
quote(x)    typeof(x)
----------  ----------
1           integer
2.71828     real
'three'     text
X'0004'     blob
NULL        null
```

You have every one of SQLite's native storage classes sitting in column x. Naturally, the question arises as to what happens when you try to sort this column. That is, before we can talk about how to sort strings, we need to review how different data types are sorted first.

When SQLite sorts a column in a result set (when it uses a comparison operator such as `<` or `>=` on the values within that column), the first thing it does is arrange the column's values according to storage class. Then within each storage class, it sorts the values according to methods specific to that class. Storage classes are sorted in the following order, from first to last:

1. `NULL` values
2. `INTEGER` and `REAL` values
3. `TEXT` values
4. `BLOB` values

Now let's modify the `SELECT` statement in the preceding example to include an `ORDER BY` clause such that the SQL is as follows:

```
select quote(x), typeof(x) from foo order by x;
```

Rerunning the query (located in `collate2.sql`) confirms this ordering:

```
C:\temp\examples\ch7> sqlite3 < collate.sql
quote(x)    typeof(x)
----------  ----------
NULL        null
1           integer
2.71828     real
'three'     text
X'0004'     blob
```

`NULL`s are first, `INTEGER` and `REAL` are next (in numerical order), `TEXT` is next, and then `BLOB`s are last.

SQLite employs specific sorting methods for each storage class. `NULL`s are obvious—there is no sort order. Numeric values are sorted numerically; integers and floats alike are compared based on their respective quantitative values. `BLOB`s are sorted by binary value. Text, finally, is where collations come in.

Collation Defined

Collation is the method by which strings are compared. A popular[1] glossary of terms defines a collation as follows:

> *Text comparison using language-sensi tive rules as o pposed to bit wise comparison of numeric character codes.*

In general, collation has to do with the ordering, comparison, and arrangement of strings and characters. Collation methods usually employ collation sequences. A collation sequence is just an ordered list of characters, usually numbered to make the order easier to remember and use. The list in turn is used to dictate how characters are ordered, compared and sorted. Given any two characters, the collation (list) can resolve which would come first or whether they are the same.

How Collation Works

Normally, a collation method compares two strings by breaking them down and comparing their respective characters using a collation. They do this by lining the strings up and comparing characters from left to right (Figure 7-2). For example, the strings `'jerry'` and `'jello'` would be compared by first

[1] See `www.ibm.com/developerworks/library/glossaries/unicode.html`.

Download from Wow! eBook <www.wowebook.com>

comparing the first letter in each string (both *j* in this case). If one letter's numerical value according to the collation sequence is larger than the other, the comparison is over: the string with the larger letter is deemed greater, and no further comparison is required. If, on the other hand, the letters have the same value, then the process continues to the second letter in each string, and to the third, and so on, until some difference is found (or no difference is found, in which case the strings are considered equal). If one string runs out of letters before the other, then it is up to the collation method to determine what this means.

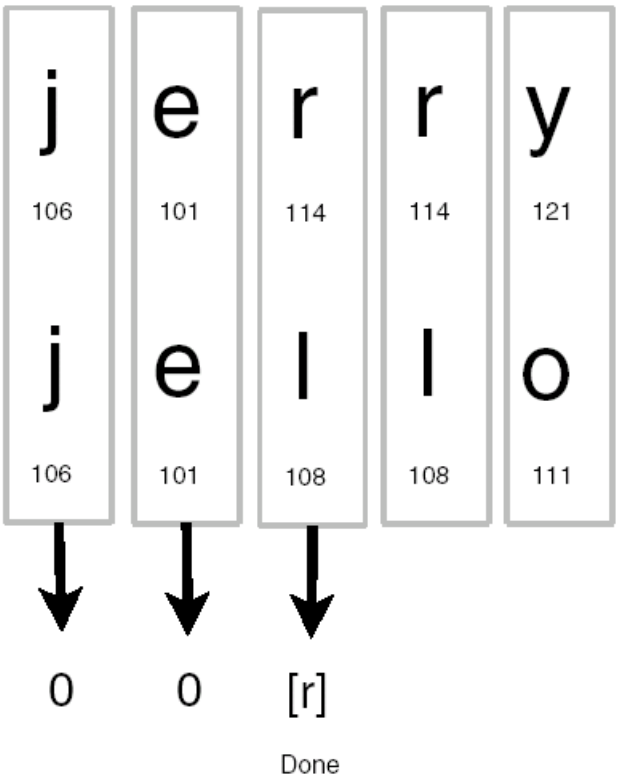


Figure 7-2. *Binary collation*

In this example, the collation is the ASCII character set. Each character's numeric ASCII value in the figure is displayed underneath it. Using the collation method just described, string comparison continues to the third character, whereupon `'jerry'` wins out, as it has an `r` with the value 114, whereas `'jello'` has the inferior `'l'` with the value 108. Based on ASCII collation, `'jerry'` > `'jello'`.

Had `'jerry'` been `'Jerry'`, then `'jello'` would have won out, because big `J`'s 74 pales to little `j`'s 106, so the comparison would have been resolved on the first character, with the prize going to `'jello'`. In short, `'jello'` > `'Jerry'`.

However, to continue the example, we could use a different collation wherein uppercase values are assigned the same numbers as their lowercase counterparts (a case-insensitive collation sequence).

Then the comparison between `'Jerry'` and `'jello'` would go back to `'Jerry'` because in this new collation `J` and `j` are the same, pulling the rug out from under `'jello'`'s lowercase `j` superiority.

But enough about collations. You get the picture. A sequence is a sequence. Ultimately, it's not the collation sequence that matters but the collation method. Collation methods don't have to use sequences. They can do whatever they want, however strange or mad that may seem.

Odds are that you are really confused by now and are just ready for a summary. So here it is. Think of collation methods as the way that strings are compared. Think of collation sequences as the way characters are compared. Ultimately, all that matters in SQLite is how strings are compared, and the two terms refer to just that. Therefore, the two terms *collation method* and *collation sequence* should just be thought of as string comparison. If you read *collation*, think string comparison. If you read *collation*, think string comparison. All in all, just think string comparison. That is the way the API is geared. When you create a custom collation, SQLite hands it two strings, *a* and *b*, and the collation returns whether string *a* is less than, equal to, or greater than string *b*. That's it, plain and simple.

Standard Collation Types

SQLite comes with a single built-in collation named `binary`, which is implemented by applying the `memcmp()` routine from the standard C library to the two strings to be compared. The `binary` collation happens to work well for English text. For other languages or locales, alternative collation may be preferred. SQLite also offers an alternate collation called `nocase`, which treats upper- and lowercase versions of the same letter as equivalents.

Collations are applied by associating them with columns in a table or an index definition or by specifying them in a query. For example, to create a case-insensitive column `bar` in `foo`, you would define `foo` as follows:

```
create table foo (bar text collate nocase, baz integer);
```

From that point on, whenever SQLite deals with a `bar`, it will use the `nocase` collation. If you prefer not to attach a collation to a database object but would rather specify it as needed on a query-by-query basis, you can specify them directly in queries, as in the following:

```
select * from foo order by bar collate nocase;
```

SQLite can also apply a collation as the result of a `cast()` expression where you convert nontext data (that wouldn't normally have a textual collation) to the `text` data type. We could cast our `baz` numeric value to text and apply the `nocase` collation like this:

```
select cast(baz) collate nocase as baz_as_text from foo order by baz_as_text;
```

Although the example is a little contrived, the capability will help you especially when dealing with data of mixed storage classes being converted to text.

A Simple Example

To jump into collations, let's begin with a simple example to get the big picture. Let's implement a collation called `length_first`. `Length_first` will take two strings and decide which one is greater depending on the length of the string. Image Jerry at the deli, demanding to only eat ingredients with short names—or Kramer insisting he has a great deal on foods with really long names. Whimsical, but very *Seinfeld*.

Collations are registered with SQLite through the `sqlite3_create_collation_v2()` function, which is declared as follows:

```
int sqlite3_create_collation_v2(
  sqlite3* db,          /* database handle */
  const char *zName,    /* collation name in SQL */
  int pref16,           /* encoding         */
  void* pUserData,      /* application data */
  int(*xCompare)(void*,int,const void*,int,const void*)
  void(*xDestroy)(void*)
);
```

It looks pretty similar to `sqlite3_create_function()`. There is the standard database handle, the collation name as it will be addressed in SQL (similar to the function/aggregate name), and the encoding, which pertains to the format of the comparison strings when passed to the comparison function.

■ **Note** Just as in aggregates and functions, separate collation comparison functions can be registered under the same collation name for each of the UTF-8, UTF-16LE, and UTF-16BE encodings. SQLite will automatically select the best comparison function for the given collation based on the strings to be compared.

The application data pointer is another API mainstay. The pointer provided there is passed on to the callback function. Basically, `sqlite3_create_collation_v2()` is a stripped-down version of `sqlite3_create_function()`, which takes exactly two arguments (both of which are text) and returns an integer value. It also has the aggregate callback function pointers stripped out.

Of the remaining two parameters, the last is `xDestroy` pointer. This function will be called at destruction time of the collation, which typically will happen when the database connection is destroyed. The destruction function will also be called if your collation is overridden by later calls to `sqlite3_create_collation_v2()`.

The Compare Function

The `xCompare` argument is the key new thing here. It points to the actual comparison function that will resolve the text values. The comparison function must have the following form:

```
int compare( void* data,        /* application data */
             int len1,          /* length of string 1 */
             const void* str1,  /* string 1 */
             int len2,          /* length of string 2 */
             const void* str2)  /* string 2 */
```

The function should return negative, zero, or positive if the first string is less than, equal to, or greater than the second string, respectively.

As stated, the `length_first` collation is a bit wishy-washy. Listing 7-5 shows its implementation.

Listing 7-5. *The* `Length_first` *Collation Function*

```
int length_first_collation( void* data, int l1, const void* s1,
                                    int l2, const void* s2 )
{
    int result, opinion;

    /* Compare lengths */
    if ( l1 == l2 ) result = 0;

    if ( l1 < l2 ) result = 1;

    if ( l1 > l2 ) result = 2;

    /* Form an opinion: is s1 really < or = to s2 ? */
    switch(result) {
        case 0: /* Equal length, collate alphabetically */
            opinion = strcmp(s1,s2);
            break;
        case 1: /* String s1 is shorter */
            opinion = -result;
            break;
        case 2: /* String s2 is shorter */
            opinion = result
            break;
        default: /* Assume equal length just in case */
            opinion = strcmp(s1,s2);
    }

    return opinion;
}
```

The Test Program

All that remains is to illustrate it in a program. Listing 7-6 shows the example program (`length_first.c`) implementation.

Listing 7-6. *The* `Length_first` *Collation Test Program*

```
int main(int argc, char **argv)
{
    char *sql; sqlite3 *db; int rc;

    sqlite3_open("foods.db", &db);

    /* Register Collation. */
    fprintf(stdout, "1. Register length_first Collation\n\n");
    sqlite3_create_collation_v2( db, "LENGTH_FIRST", SQLITE_UTF8, db,
                              length_first_collation, length_first_collation_del );
```

```
    /* Turn SQL logging on. */
    log_sql(db, 1);

    /* Test default collation. */
    fprintf(stdout, "2. Select records using default collation.\n");
    sql = "select name from foods order by name";
    print_sql_result(db, sql);

    /* Test Length First collation. */
    fprintf(stdout, "\nSelect records using length_first collation. \n");
    sql = "select name from foods order by name collate LENGTH_FIRST";
    print_sql_result(db, sql);

    /* Done. */
    sqlite3_close(db);

    return 0;
}
```

Results

Running the program yields the following results (abbreviated here to save space):

```
1. Register length_first Collation

2. Select records using default collation.
  TRACE: select name from foods order by name
name
-----------------
A1 Sauce
All Day Sucker
Almond Joy
Apple
Apple Cider
Apple Pie
Arabian Mocha Java (beans)
Arby's Roast Beef
Artichokes
Atomic Sub
...
```

```
Select records using length_first collation.
  TRACE: select name from foods order by name collate LENGTH_FIRST
issue
-----------------
BLT
Gum
Kix
Pez
Pie
Tea
Bran
Duck
Dill
Life
...
```

Our program starts by registering the `length_first` collation sequence. We then proceed to select records using SQLite's default `binary` collation, which returns records in alphabetical order. Then comes our use of the `length_first` collation, where we select records again. As the results indicate, we are right on the mark, sorting first by length and then alphabetically.

Collation on Demand

SQLite provides a way to defer collation registration until it is actually needed. So if you are not sure your application is going to need something like the `length_first` collation, you can use the `sqlite3_collation_needed()` function to defer registration to the last possible moment (perhaps as the result of some secret last-minute food renaming). You simply provide `sqlite3_collation_needed()` with a callback function, which SQLite can rely on to register unknown collation sequences as needed, given their name. This acts like a callback for a callback, so to speak. It's like saying to SQLite, "Here, if you are ever asked to use a collation that you don't recognize, call this function, and it will register the unknown sequence; then you can continue your work."

The `sqlite3_collation_needed()` function is declared as follows:

```
int sqlite3_collation_needed(
  sqlite3* db,    /* connection handle  */
  void* data,     /* application data   */
  void(*crf)(void*,sqlite3*,int eTextRep,const char*)
);
```

The `crf` argument (the collation registration function as we call it) points to the function that will register the unknown collation. For clarity, it is defined as follows:

```
 void crf( void* data,    /* application data */
           sqlite3* db,   /* database handle */
           int eTextRep,  /* encoding */
           const char*)   /* collation name */
```

The `db` and `data` arguments of `crf()` are the values passed into the first and second arguments of `sqlite3_collation_needed()`, respectively.

Summary

By now you've discovered that user-defined functions, aggregates, and collations can be surprisingly useful. The ability to add such functions using the extensions part of the C API is a great complement to the open nature of the core SQLite library. So, you're free to dig in and modify SQLite to your heart's content. We would even go so far to say that SQLite provides a friendly, easy-to-user interface that makes it possible to implement a wide range of powerful extensions and customizations, especially when combined with other features already present in SQLite.

In the next chapter, you will see how many extension languages take advantage of this. They use the C API to make it possible to implement all manner of capabilities and extensions for a given language.

CHAPTER 8

Language Extensions

SQLite is written in C and has its own C API, which makes C the "native language" to some degree. The open source community, however, has provided many extensions for SQLite that make it accessible to many programming languages and libraries such as Perl, Python, Ruby, Java, .NET/C#, Qt, and ODBC. In many cases, there are multiple extensions to choose from for a given language, developed by different people to meet different needs.

Many extensions conform to various API standards. For instance, one of the SQLite Perl extensions follows the Perl DBI—Perl's standard interface for database access. Similarly, one of the Python extensions conforms to Python's DB API specification, as does at least one of the Java extensions to JDBC. Regardless of their particular APIs, internally all extensions work on top of the same SQLite C API, which you've already explored in Chapters 5, 6, and 7. To some degree, all extensions reflect the C foundation of the SQLite design. Some extensions provide both a standard interface that conforms to their particular API standard as well as an alternative interface that better reflects the design of the SQLite API. Therefore, in such cases, you can choose which interface works best for you based on your requirements.

In general, all extensions have the same comparative anatomy regardless of their particular API. They follow a similar pattern. Once you understand this pattern, every interface will start to look similar in many respects. There is some kind of connection object representing a single connection to the database, and some kind of cursor object representing a SQL query from which you can both execute commands and iterate over results. Conceptually, it is quite simple. Internally, you are looking at a `sqlite3` structure and a `sqlite3_stmt` structure, both well-known structures from our earlier explanation and examples. The same rules apply for language extensions as for the C API.

This chapter covers language extensions for six popular languages: Perl, Python, Ruby, Java, Tcl, and PHP. The intent is to provide you with a convenient introduction with which to quickly get started using SQLite in a variety of different languages. We'll highlight right now that we won't be teaching you those languages themselves—so some knowledge of Perl, or Python, or Ruby, and so on, is expected.

Coverage for each language follows a common outline composed of the common topics of questions from people starting out with language extensions for SQLite:

- Connecting to databases
- Executing queries
- Using bound parameters
- Implementing user-defined functions or aggregates

Together these topics constitute the vast majority of what you will ever use in any particular SQLite extension. You might find some "developer tension" over developing user-defined functions in SQLite versus implementing the logic in your chosen language. Although we can't make that decision for you, we can give you the knowledge and expertise to help make that choice yourself.

Not every extension supports all of these topics. For instance, some don't offer bound parameters. Conversely, some extensions offer unique features that are outside the topics covered here. The aim here is to provide a consistent, straightforward process with which to easily get started using SQLite in a wide variety of languages. Once you get started with any extension, it is usually easy to pick up any special features available in that extension.

Where appropriate, this chapter addresses how different extensions use various parts of the C API in order to help you understand what is going on under the hood. If you have not read the preceding chapters on the C API, we strongly recommend that you do so. No matter what language you program in, it will almost always be helpful to understand how SQLite works in order to make the most of it and thus write good code. It will help you select the most suitable query methods, understand the scope of transactions, and know how to anticipate, deal with, or avoid locks, among other things.

Selecting an Extension

For many of today's popular languages, you are spoilt for choice when it comes to language bindings for SQLite. Choosing the extension that suits you is often more than a straight technical decision. To help, we cover the interface that best fits the following criteria:

- Support for the latest SQLite 3 versions
- Good documentation
- Stability
- Portability

Despite these criteria, there were multiple candidates in some cases that met all the qualifications. In such cases, the tiebreaker was more a matter of personal preference than anything else. But the purpose here is not to favor a particular extension. It is to teach concepts so that you can easily pick any extension in any language and quickly put it and SQLite to good use.

The reasons we chose a particular extension may not be the reasons you would. Some additional points to consider include the following:

- *License and license compatibility*: The extension's license can directly affect how you can use it. Is it open source, and if so, under which license? If you are writing code for a commercial product, you definitely need to take this into consideration.
- *Data type mapping*: How does the extension map SQLite's storage classes to the language's native types. Are all values returned as text? Is some kind of crazy casting scheme used? If not, is there an easy way to determine the mappings? Do you have any control over how the mapping is done?
- *Query methods*: Three different query methods are supported in the C API. Which one does the extension use? Ideally it supports all three.
- *API coverage*: How well does the extension cover other areas that don't easily map to many standard database interfaces? For example, does it allow you to call the SQL trace function or operational security functions? Do you actually need these?

Download from Wow! eBook <www.wowebook.com>

- *Linkage and distribution*: How easy is it to use the extension with a particular version of SQLite? Does it include a version of SQLite in the distribution? Does it use a shared library, or is it statically linked to a particular version? What if you need to upgrade SQLite; how easily can you do this with the extension? Will anything break?

There are other considerations as well, such as community support, mailing list activity, maintainer support and responsiveness, regression testing, code quality...the list goes on. All of these things may play an important role in your decision to use a particular extension. Only you can answer these questions for yourself.

The SQLite wiki has a page that provides an exhaustive list of language interfaces, located at `www.sqlite.org/cvstrac/wiki?p=SqliteWrappers`. All of the interfaces covered here were taken from this list. The source code for all of the examples in this chapter is located in the `ch8` directory of the `examples` zip file, available at the Apress web site.

Perl

There are two SQLite extensions for Perl. The first is a modeled on the Perl DBI standard for database connectivity; therefore, if you understand DBI, you will have no trouble using the extension. For those unfamiliar with the Perl DBI, you can review full documentation on the CPAN web site at `http://search.cpan.org/search?module=DBI`. The second extension is a SWIG-based wrapper for use with Perl. We'll examine the DBI interface in the following sections.

Installation

The current version of the SQLite DBI driver at the time of this writing is `DBD-SQLite-1.29`. You can install it by using CPAN or by manually building the package from source. The prerequisites for the SQLite DBD module are, of course, Perl, a C compiler, and the DBI module. The latest versions of the SQLite DBI driver also require YAML ("Yet another markup language" or "YAML Ain't Markup Language," depending on whom you believe), used for various configuration files. To install `DBD::SQLite` using CPAN, invoke the CPAN shell from the command line as follows:

```
fuzzy@linux $ cpan

cpan> install DBD::SQLite

... lots of CPAN install messages ...
```

To install from source in Linux/Unix, change to a temporary directory and do the following:

```
fuzzy@linux $ wget http://search.cpan.org/CPAN/authors/id/A/AD/ADAMK/DBD-SQLite-1.29.tar.gz

... lots of wget messages ...

fuzzy@linux $ tar xzvf DBD-SQLite-1.29.tar.gz

... lots of tar messages ...

fuzzy@linux $ cd DBD-SQLite-1.29
```

```
fuzzy@linux DBD-SQLite-1.29 $ perl Makefile.PL

... lots of perl messages ...

Checking installed SQLite version...
Looks good
fuzzy@linux DBD-SQLite-1.29 $ make install

... more perl messages ...
```

The SQLite Perl extension includes its own copy of the SQLite3 binaries, so there is no need to compile and install SQLite beforehand. SQLite is embedded in the extension.

Connecting

To check the SQLite driver is installed, use the DBI function `available_drivers()` within Perl:

```
use DBI;
print "Drivers: " . join(", ", DBI->available_drivers()), "\n";
```

You connect to a database using the standard `DBI::connect` function, as follows:

```
use DBI;
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });
$dbh->disconnect;
```

The second and third arguments correspond to the driver name and database name. The third and fourth correspond to username and password, which are not applicable to SQLite.

The function returns a database handle object representing the database connection. Internally, this corresponds to a single `sqlite3` structure. You can create in-memory databases by passing `:memory:` for the name of the database. In this case, all tables and database objects will then reside in memory for the duration of the session and will be destroyed when the connection is closed. You close the database using the database handle's `disconnect` method.

Query Processing

Queries are performed using the standard DBI interface as well. You can use the `prepare()`, `execute()`, `fetch()`, or `selectrow()` functions of the database connection handle. All of these are fully documented in the DBI man page. Listing 8-1 shows an example of using queries.

Listing 8-1. *Executing Queries in Perl*

```
use DBI;
# Connect to database
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });
```

```
# Prepare the statment
my $sth = $dbh->prepare("select name from foods limit 3");

# Execute
$sth->execute;

# Print a human-readable header
print "\nArray:\n\n";

# Iterate over results and print
while($row = $sth->fetchrow_arrayref) {
  print @$row[0] . "\n";
}

# Do the same thing, this time using hashref
print "\nHash:\n\n";
$sth->execute;
while($row = $sth->fetchrow_hashref) {
  print @$row{'name'} . "\n";
}

# Finalize the statement
$sth->finish;

#Disconnect
$dbh->disconnect;
```

Internally, the prepare() method corresponds to the `sqlite3_prepare_v2()` method. Similarly, `execute()` calls the first `sqlite3_step()` method. It automatically figures out whether there is data to be returned. If there is no data (for example, non-`select` statements), it automatically finalizes the query. The `fetchrow_array()`, `fetchrow_hashref()`, and `fetch()` methods call `sqlite3_step()` as well, returning a single row in the result set until the results are exhausted.

Additionally, you can run non-`select` statements in one step using the database object's `do()` method. Listing 8-2 illustrates this using an in-memory database.

Listing 8-2. *The `do()` Method*

```
use DBI;
my $dbh = DBI->connect("dbi:SQLite:dbname=:memory:", "", "", { RaiseError => 1 });
$dbh->do("create table cast (name)");
$dbh->do("insert into cast values ('Elaine')");
$dbh->do("insert into cast values ('Jerry')");
$dbh->do("insert into cast values ('Kramer')");
$dbh->do("insert into cast values ('George')");
$dbh->do("insert into cast values ('Newman')");

my $sth = $dbh->prepare("select * from cast");
$sth->execute;
```

```
while($row = $sth->fetch) {
    print join(", ", @$row), "\n";
}
$sth->finish;
$dbh->disconnect;
```

The statement handle object's `finish()` method will call `sqlite3_finalize()` on the query object, if it has not already done so. This program produces the following output:

```
Elaine
Jerry
Kramer
George
Newman
```

Parameter Binding

Parameter binding follows the method defined in the DBI specification. Although SQLite supports both positional and named parameters, the Perl interface uses only positional parameters, demonstrated in Listing 8-3.

Listing 8-3. *Parameter Binding in Perl*

```
use DBI;

my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

my $sth = $dbh->prepare("select * from foods where name like :1");
$sth->execute('C%');

while($row = $sth->fetchrow_hashref) {
    print @$row{'name'} . "\n";
}

$sth->finish;
$dbh->disconnect;
```

The `execute()` method takes the parameter values defined in `prepare()`. Running this against the evolving foods database we've used for many of our examples returns 73 rows (depending on how much fun you had playing with chocolate bobka in previous chapters).

User-Defined Functions

User-defined functions and aggregates are implemented using private methods of the driver. They follow closely to their counterparts in the SQLite C API. Functions are registered using the following private method:

```
$dbh->func( $name, $argc, $func_ref, "create_function" )
```

Here `$name` specifies the SQL function name, `$argc` specifies the number of arguments, and `$func_ref` specifies a reference to the Perl function that provides the implementation. Listing 8-4 illustrates an implementation of `hello_newman()` in Perl.

Listing 8-4. *`hello_newman()` in Perl*

```
use DBI;

sub hello_newman {
    return "Hello Jerry";
}

# Connect
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# Register function
$dbh->func('hello_newman', 0, \&hello_newman, 'create_function');

# Call it
print $dbh->selectrow_arrayref("select hello_newman()")->[0] . "\n";

$dbh->disconnect;
```

If you provide -1 as the number of arguments, then the function will accept a variable number of arguments.

Aggregates

User-defined aggregates are implemented as packages. Each package implements a step function and a finalize function. The step function is called for each row that is selected. The first value supplied to the function is the context, which is a persistent value maintained between calls to the function. The remaining values are the arguments to the aggregate function supplied in the SQL statement. The finalize function is provided the same context as the step function. The following example implements a simple `SUM` aggregate called `perlsum`. The aggregate is implemented in a package called `perlsum.pm`, shown in Listing 8-5.

Listing 8-5. *The `perlsum()` Aggregate*

```
package perlsum;

sub new { bless [], shift; }
```

```
sub step {
<A HREF="http://rubyforge.org/projects/sqlite-ruby" CLASS="URL">    my ( $self, $value ) =
@_;
    @$self[0] += $value;
}

sub finalize {
    my $self = $_[0];
    return @$self[0];
}

sub init {
    $dbh = shift;
    $dbh->func( "perlsum", 1, "perlsum", "create_aggregate" );
}

1;
```

The `init` function is used to register the aggregate in the database connection.

Python

There are numerous approaches to working with SQLite in Python. These can be grouped into two camps. The first is the group of SQLite wrappers, such as PySQLite and APSW, that act as straightforward wrapper interfaces to the underlying SQLite API calls (though with varying degrees of depth). The second group is the all-encompassing framework approach, such as SQL Alchemy. These are typically designed as complete object-relational mapping systems, seeking not only to provide access to the underlying database API but to also follow object-oriented philosophies in the design and use of database-related code.

At last count, there were at least nine different wrappers or object-relational frameworks for SQLite in Python. That's almost enough for a book in its own right. We'll cover one of the more popular wrappers, PySQLite, principally because it has been one of the most enduring and also maps to the Python DB API.

Michael Owens (the author of the first edition of this book) wrote the original version of PySQLite and started the project in 2002, which then was written using the SQLite 2.*x* API. Gerhard Häring has since taken over the project and completely rewritten it for PySQLite version 2, which supports SQLite version 3. The project has also evolved and moved several times and now refers to itself as pysqlite (all lowercase). You can find project information for PySQLite on Google Code at `http://code.google.com/p/pysqlite/`. Both the source code and native Windows binaries are available for download.

Installation

From Python 2.5 onward, pysqlite is included as a standard module for Python and is simply referred to as the sqlite3 module when used from the standard library. There are still some uses to separately acquiring and compiling/using the pysqlite stand-alone module, because it tracks bug fixes and new features more closely than the Python distribution itself.

To source your own pysqlite package, Windows users are encouraged to use the provided binaries rather than building them by hand. On Linux, Unix, Mac OS X, and other systems, you build and install pysqlite using Python's `distutils` package, or you source the rpm or deb file from your local package

repository. The prerequisites for compiling your own PySQLite are a C compiler, Python 2.3 or newer, and SQLite 3.1 or newer. After unpacking the source code, compile the code with the following command:

```
python setup.py build
```

`distutils` should normally be able to figure out where everything is on your system. If not, you will get an error, in which case you need to edit the `setup.cfg` file in the main directory to point out where things are. If everything goes well, you should see "Creating library..." in the output, indicating a successful build. Next install the library:

```
python setup.py install
```

You should not see any errors. If you do, report them to the community through the quite active group at `http://groups.google.com/group/python-sqlite`.

For Windows systems, simply download the binary distribution and run the installer. The only prerequisite in this case is that SQLite 3.2 or newer be installed on your system.

Connecting

As mentioned, the built-in module in Python 2.5 and newer is named sqlite3. If using the separately sourced pysqlite package, it is in `pysqlite2`. The DB API module is located in `dbapi2`. This is sometimes confusing when you are thinking of PySQLite version numbers and SQLite version numbers simultaneously. You connect to a database using the module's `connect()` function, passing in the relative name or complete file path of the database file you want to open.

For the built-in pysqlite in Python 2.5+, a connection would be as follows:

```
import sqlite3

db = sqlite3.connect("foods.db")
```

With the separately sourced package for pysqlite, your connect changes to the following:

```
from pysqlite2 import dbapi2 as sqlite3

db = sqlite3.connect("foods.db")
```

The usual rules apply: you can use in-memory databases by passing in the name `:memory:` as the database name; new databases are created for new files; and so forth. You can see the only real variation in the included module and separately sourced module for pysqlite is in import semantics. From here on, we'll assume you're happy switching to the technique that best suits your preferred way of using pysqlite.

Query Processing

Query execution is done according to the Python DB API Specification 2. You create a cursor object with which to run the query and obtain results. Internally, a cursor object holds a `sqlite3_stmt` handle. Listing 8-6 illustrates executing a basic query.

Listing 8-6. *A Basic Query in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite3

# Connect
con = sqlite3.connect("foods.db")

# Prepare a statement and execute. This calls sqlite3_prepare() and sqlite3_step()
cur = con.cursor()
cur.execute('select * from foods limit 10')

# Iterate over results, print the name field (row[2])
row = cur.fetchone()
while row:
    print row[2]
    # Get next row
    row = cur.fetchone()

cur.close()
con.close()
```

Remember, Python code is white-space sensitive, so be careful not to introduce additional spaces or tabs when copying this code. Running this code produces the following output:

```
Bagels
Bagels, raisin
Bavarian Cream Pie
Bear Claws
Black and White cookies
Bread (with nuts)
Butterfingers
Carrot Cake
Chips Ahoy Cookies
Chocolate Bobka
```

Query compilation and the first step of execution are performed together in the cursor's `execute()` method. It calls `sqlite3_prepare()` followed by `sqlite3_step()`. For queries that modify data rather than return results, this completes the query (short of finalizing the statement handle). For `select` statements, it fetches the first row of the result. The `close()` method finalizes the statement handle.

PySQLite 2 supports iterator-style result sets, similar to other Python database wrappers. For example, Listing 8-6 could be rewritten as shown in Listing 8-7.

Listing 8-7. *Using Pythonic Iterators for a Query*

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()
cur.execute('select * from foods limit 10')
```

```
for row in cur:
    print row[2]
```

Parameter Binding

PySQLite supports parameter binding by both position and name. This can also be done using the cursor's `execute()` method, which we introduced in Listing 8-6. Compilation, binding, and the first step of execution are performed in the one call to `execute()`. You specify positional parameters by passing a tuple as the second argument to `execute()`. You specify named parameters by passing a dictionary rather than a tuple. Listing 8-8 illustrates both forms of parameter binding.

Listing 8-8. *Parameter Binding in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()
cur.execute('insert into episodes (name) values (?)', ('Soup Nazi'))
cur.close()

cur = con.cursor()
cur.execute('insert into episodes (name) values (:name)', {'name':'Soup Nazi'})
cur.close()
```

This model does not support the reuse of compiled queries (`sqlite3_reset()`). To do this, use `Cursor.executemany()`. While the DB API specifies that `executemany` should take a list of parameters, PySQLite extends it to accommodate iterators and generators as well. For example, the canonical form is shown in Listing 8-9.

Listing 8-9. *The `executemany()` Method*

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()

episodes = ['Soup Nazi', 'The Fusilli Jerry']
cur.executemany('insert into episodes (name) values (?)', episodes)
cur.close()
con.commit()
```

But you could just as easily use a generator, as shown in Listing 8-10.

Listing 8-10. *The* `executemany()` *Method with a Generator*

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()

def episode_generator():
    episodes = ['Soup Nazi', 'The Fusilli Jerry']
    for episode in episodes:
        yield (episode,)

cur.executemany('INSERT INTO episodes (name) VALUES (?)', episode_generator())
cur.close()
con.commit()
```

In both Listing 8-9 and Listing 8-10, we use the power of parameter binding with a single compilation of the query in question. The query is reused for each item in the sequence or iterator, improving the overall performance for such batches of identical queries.

Download from Wow! eBook <www.wowebook.com>

TRANSACTION HANDLING IN PYSQLITE

It is important to notice the `con.commit()` lines in the previous examples. Don't forget them if you use PySQLite. If you don't include them, then all transactions in the examples will be rolled back. This behavior differs from both the SQLite C API and other extensions, and it might surprise you. By default, SQLite runs in autocommit mode. PySQLite, on the other hand, starts and finishes transactions behind the scenes, using its own logic for starting and committing transactions based on what kind of SQL you execute. Before passing your SQL statement to SQLite, it is analyzed by PySQLite. If it finds an `insert`, `update`, `delete`, or `replace`, PySQLite will implicitly start a transaction (issue a `begin`). Then, if you issue any other kind of command following it, PySQLite will automatically issue a `commit` before executing that command. This transaction logic places an additional constraint on your code in that you cannot use `ON CONFLICT ROLLBACK`; otherwise, it interferes with PySQLite's monitoring of your transactions.

It is possible to turn off this behavior and restore the default SQLite behavior. You can explicitly turn autocommit mode back on by setting `isolation_level` of the database connection to `None`, as follows:

```
from pysqlite2 import dbapi2 as sqlite3

# Turn on autocommit mode
con = sqlite3.connect("foods.db", isolation_level=None)
```

User-Defined Functions

You register user-defined functions using `create_function()`, which takes the following form:

```
con.create_function(name, args, pyfunc)
```

Here, `name` is the SQL name of the function, `args` is the number of arguments accepted by the function, and `pyfync` is the Python function that implements the SQL function. Listing 8-11 shows a Python implementation of `hello_newman()` using PySQLite.

Listing 8-11. *`hello_newman()` in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite3

# The Python function
def hello_newman():
    return 'Hello Jerry'

con = sqlite3.connect(":memory:")
con.create_function("hello_newman", 0, hello_newman)
cur = con.cursor()

cur.execute("select hello_newman()")
print cur.fetchone()[0]
```

If you use -1 as the number of arguments, then the function will accept a variable number of arguments.

Aggregates

PySQLite implements user-defined aggregates as Python classes. These classes must implement `step()` and `finalize()` methods. You register aggregates using `Connection::create_aggregate()`, which takes three arguments: the function's SQL name, the number of arguments, and the class name that implements the aggregate. Listing 8-12 implements a simple `SUM` aggregate called `pysum`.

Listing 8-12. *The `pysum()` Aggregate in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite3

class pysum:
    def init(self):
        self.sum = 0

    def step(self, value):
        self.sum += value

    def finalize(self):
        return self.sum
```

```
con = sqlite3.connect("foods.db")
con.create_aggregate("pysum", 1, pysum)
cur = con.cursor()
cur.execute("select pysum(id) from foods")
print cur.fetchone()[0]
```

APSW as an Alternative Python Interface

APSW is written and maintained by Roger Binns. Detailed information on APSW can be found at `http://code.google.com/p/apsw/`. Although APSW and pysqlite share some common capabilities and a very similar syntactical style, APSW goes further in providing a range of additional features and API mapping points. Rather than repeating very similar Python code samples, we'll outline the key benefits of APSW and allow you to explore them as you want.

Support for virtual tables, VFS, blob I/O, backups, and file control: APSW provides comprehensive support for the latest features provided by SQLite, as well as database management tasks that aren't provided through pysqlite's narrower focus on DBAPI conformity.

APSW Python Shell: APSW incorporates an extensible shell that can be used in place of the SQLite command interface. This includes great features such as color-coded syntax, CSV formatting and handling, strictly correct error handling, and additional formatting options for data and messages.

Consistent transaction handling: APSW always performs transactions as you would expect, rather than mimicking pysqlite's attempt to intuitively handle them without your involvement.

There are also a host of other nice little (and even major) features that make APSW a joy to work with. You can review all of these capabilities at `http://apidoc.apsw.googlecode.com/hg/pysqlite.html`.

Ruby

The Ruby extension was written by Jamis Buck of 37 Signals and is now maintained by a group of developers. You can obtain detailed information on the extension as well as the source code at `http://rubyforge.org/projects/sqlite-ruby`. The complete documentation for the SQLite 3 extension can be found at `http://sqlite-ruby.rubyforge.org/sqlite3/`, though the layout is a little daunting.

Installation

The extension can be built in two ways: with or without SWIG. The Ruby configuration script will automatically figure out whether SWIG is available on your system. You should build the extension with SWIG (the C extension), because that implementation is more stable. If your operating system uses a well-developed package management and repository system (like Debian, Ubuntu, and so on), you'll also find packages available for your system there.

At the time of this writing, the current version of `sqlite-ruby` is 1.3.1. To build from source, fetch the tarball from `http://rubyforge.org/projects/sqlite-ruby`. Unpack the tarball, and run three setup commands:

```
fuzzy@linux $ tar xjvf sqlite3-ruby-1.3.1.tar.bz2
fuzzy@linux $ cd sqlite3-ruby-1.3.1
fuzzy@linux $ ruby setup.rb config
fuzzy@linux $ ruby setup.rb setup
fuzzy@linux $ ruby setup.rb install
```

If you have Ruby gems installed, you can get Ruby to do everything for you in one step:

```
fuzzy@linux $ gem install --remote sqlite3-ruby
```

If you are installing from a repository, ensure you check your installed version of Ruby and match the right version from the repository. Because old versions of SQLite also enjoy support, ensure you install the libsqlite3-ruby or equivalent package for your platform. For example, under Ubuntu:

```
fuzzy@linux $ ruby --version
ruby 1.8.6 (2007-09-24 patchlevel 111) [x86_64-linux]
fuzzy@linux $ sudo apt-get install libsqlite3-ruby libsqlite-ruby libsqlite-ruby1.8
... many apt messages fly past ...
```

Connecting

To load the SQLite extension, you must import the sqlite module, using either `load` or `require`, as follows:

```
require 'sqlite'
```

You connect to a database by instantiating a `SQLite::Database` object, passing in the name of the database file. By default, columns in result sets are accessible by their ordinal. However, they can be accessed by column name by setting `Database::results_as_hash` to `true`:

```
require 'sqlite'
db = SQLite::Database.new("foods.db")
db.results_as_hash = true
```

Query Processing

The Ruby extension follows the SQLite API quite closely. It offers both prepared queries and wrapped queries.

Prepared queries are performed via `Database::prepare()`, which passes back a `Statement` object, which holds a `sqlite3_stmt` structure. You execute the query using `Statement`'s `execute` method, which produces a `ResultSet` object. You can pass the `Statement` a block, in which case it will yield the `ResultSet` object to the block. If you don't use a block, then the `Statement` will provide the `ResultSet` object as a return value. `ResultSet` is used to iterate over the returned rows. Internally, it uses `sqlite3_step()`. You can get at the rows in `ResultSet` either through a block using the `each()` iterator or by using a conventional loop with the `next()` method, which returns the next record in the form of an array or `nil` if it has reached the end of the set. Listing 8-13 shows prepared queries in action in Ruby.

Listing 8-13. *Prepared Queries in Ruby*

```
#!/usr/bin/env ruby

require 'sqlite3'

db = SQLite3::Database.new('foods.db')

stmt = db.prepare('select name from episodes')

stmt.execute do | result |
  result.each do | row |
    puts row[0]
  end
end

result = stmt.execute()
result.each do | row |
  puts row[0]
end

stmt.close()
```

It is important to call `Statement.close()` when you are done to finalize the query. Because a `Statement` object holds a `sqlite3_stmt` structure internally, each subsequent call to `execute` thus reuses the same query, avoiding the need to recompile the query.

Parameter Binding

The Ruby extension supports both positional and named parameter binding. You bind parameters using `Statement::bind_param()` and/or `Statement::bind_params()`. `bind_param()` has the following form:

```
bind_param(param, value)
```

If `param` is a `Fixnum`, then it represents the position (index) of the parameter. Otherwise, it is used as the name of the parameter. `bind_params()` takes a variable number of arguments. If the first argument is a hash, then it uses it to map parameter names to values. Otherwise, it uses each argument as a positional parameter. Listing 8-14 illustrates both forms of parameter binding.

Listing 8-14. *Parameter Binding in Ruby*

```
require 'sqlite3'

db = SQLite3::Database.new("foods.db")
db.results_as_hash = true

# Named paramters
```

```
stmt = db.prepare('select * from foods where name like :name')
stmt.bind_param(':name', '%Peach%')

stmt.execute() do |result|
  result.each do |row|
    puts row['name']
  end
end

# Positional paramters

stmt = db.prepare('select * from foods where name like ? or type_id = ?')
stmt.bind_params('%Bobka%', 1)

stmt.execute() do |result|
  result.each do |row|
    puts row['name']
  end
end

# Free read lock
stmt.close()
```

If you don't need to use parameters, a shorter way to process queries is using `Database::query()`, which cuts out the `Statement` object and just returns a `ResultSet`, as shown in Listing 8-14.

Listing 8-14. *Using the* `Database::query()` *Method in Ruby*

```
require 'sqlite3'

db = SQLite3::Database.new("foods.db")
db.results_as_hash = true

result = db.query('select * from foods limit 10')
result.each do |row|
  puts row['name']
end

result.reset()

while row = result.next
  puts row['name']
end
result.close()
```

Like `Statement` objects, `Result` objects are also thin wrappers over statement handles and therefore represent compiled queries. They can be rerun with a call to `reset`, which calls `sqlite3_reset()` internally and reexecutes the query. Unlike `Statement` objects however, they cannot be used for bound parameters.

Other, even shorter, query methods include `Database::get_first_row()`, which returns the first row of a query, and `Database::get_first_value()`, which returns the first column of the first row of a query.

User-Defined Functions

User-defined functions are implemented using `Database::create_function()`, which has the following form:

```
create_function( name, args, text_rep=Constants::TextRep::ANY ) {|func, *args| ...}
```

Here, `name` is the name of the SQL function, `args` is the number of arguments (-1 is variable), and `text_rep` corresponds to the UTF encoding. Values are `UTF8`, `UTF16LE`, `UTF16BE`, `UTF16`, and `ANY`. Finally, the function implementation is defined in the block. Listing 8-15 illustrates a Ruby implementation of `hello_newman()`.

Listing 8-15. *`hello_newman()` in Ruby*

```
require 'sqlite3'

db = SQLite3::Database.new(':memory:')

db.create_function('hello_newman', -1 ) do |func, *args|
  if args.length == 0
    func.result = 'Hello Jerry'
  else
    func.result = 'Hello %s' % [args.join(', ')]
  end
end

puts db.get_first_value("SELECT hello_newman()")
puts db.get_first_value("SELECT hello_newman('Elaine')")
puts db.get_first_value("SELECT hello_newman('Elaine', 'Jerry')")
puts db.get_first_value("SELECT hello_newman('Elaine', 'Jerry', 'George')")
```

This program produces the following output:

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

Java

Like Python mentioned earlier, a wealth of Java extensions are available for SQLite. These fall into two broad camps: the JDBC drivers (sometimes with added goodies) and complete reimplementations of SQLite in Java. We'll cover the javasqlite wrapper written by Christian Werner, who wrote the SQLite ODBC driver as well. It includes both a JDBC driver and a native JNI extension, which closely shadows the SQLite C API. You can download the extension from `www.ch-werner.de/javasqlite/overview-summary.html#jdbc_driver`.

The main class in the JNI extension is SQLiteDatabase. Most of its methods are implemented using callbacks that reference the following interfaces:

- `SQLite.Callback`
- `SQLite.Function`
- `SQLite.Authorizer`
- `SQLite.Trace`
- `SQLite.ProgressHandler`

The `SQLite.Callback` interface is used to process result sets through row handlers, as well as column and type information. SQLite.Authorizer is a thin wrapper over the SQLite C API function `sqlite3_set_authorizer()`, with which you can intercept database events before they happen. `SQLite.Trace` is used to view SQL statements as they are compiled, and it wraps `sqlite3_trace()`. `SQLite.ProgressHandler` wraps `sqlite3_progress_handler()`, which is used to issue progress events after a specified number of VDBE instructions have been processed.

Installation

The current version requires JDK 1.4 or newer. The extension uses GNU Autoconf, so building and installing requires only three steps:

```
fuzzy@linux $ tar xvzf javasqlite-20100727.tar.gz
fuzzy@linux $ ./configure
fuzzy@linux $ make
fuzzy@linux $ make install
```

The configure script will look for SQLite and the JDK in several default locations. However, to explicitly specify where to look for SQLite and the JDK, several command-line options are available.

`--with-sqlite=DIR` to specify the sqlite v2 location

`--with-sqlite3=DIR` to specify the sqlite v3 location

`--with-jdk=DIR` to specify the JDK location.

To specify the place where the resulting library should be installed (`libsqlite_jni.so` file), use the `--prefix=DIR` option. The default location is `/usr/lib/jvm/java-6-sun/jre/lib/i386` (though this may vary depending on your distribution of Linux or use of another Unix flavor or Mac OS X, such as `/usr/local/lib`, for example.) To specify where the `sqlite.jar` is to be installed, use the `--with-jardir=DIR` option. The default is `/usr/lib/jvm/java-6-sun-1.6.0.20/jre/lib/ext/sqlite.jar` (again, this may vary depending on the age and type of your distribution). This file contains the high-level part and the JDBC driver. At runtime, it is necessary to tell the JVM both places with the `-classpath` and `-Djava.library.path=..` command-line options.

For Windows, the makefiles `javasqlite.mak` and `javasqlite3.mak` are provided in the distribution. They contain some build instructions and use the J2SE 1.4.2 from Sun and Microsoft Visual C++ 6.0. A DLL with the native JNI part (including SQLite 3.7.0) and the JAR file with the Java part can be downloaded from the web site.

Connecting

You connect to a database using `SQLiteDatabase::open()`, which takes the name of the database and the file mode, as shown in Listing 8-16. The file mode is an artifact from SQLite 2 API and is no longer used. You can provide any value to satisfy the function. The code in Listing 8-16 is taken from the example file `SQLiteJNIExample.java`. The full code from this example illustrates all the main facets of database operations: connecting, querying, using functions, and so forth.

Listing 8-16. *The JavaSQLite Test Program*

```
import SQLite.*;

public class SQLiteJNIExample
{
  public static void main(String args[])
  {
    SQLite.Database db = new SQLite.Database();

    try
      {
        db.open("foods.db", 700);

        // Trace SQL statements
        db.trace(new SQLTrace());

        // Query example
        query(db);

        // Function example
        user_defined_function(db);

        // Aggregate example
        user_defined_aggregate(db);

        db.close();
      }
      catch (java.lang.Exception e)
      {
        System.err.println("error: " + e);
      }
  }

  ...
}
```

Query Processing

Your queries are issued using `SQLiteDatabase::compile()`. This function can process a string containing multiple SQL statements. It returns a `VM` (virtual machine) object that holds all the statements.

The VM object parses each individual SQL statement on each call to compile(), returning true if a complete SQL statement was compiled and false otherwise. You can therefore iterate through all SQL statements in a loop, breaking when the VM has processed the last statement.

When the VM has compiled a statement, you can execute it using VM::step(). This function takes a single object, which implements a SQLite.Callback interface. The example uses a class called Row for this purpose, which is shown in Listing 8-17.

Listing 8-17. *The Row Class*

```
class Row implements SQLite.Callback
{
  private String row[];

  public void columns(String col[]) {}
  public void types(String types[]) {}

  public boolean newrow(String data[])
  {
    // Copy to internal array
    row = data;
    return false;
  }

  public String print()
  {
    return "Row:    [" + StringUtil.join(row, ", ") + "]";
  }
}
```

The SQLite.Callback interface has three methods: columns(), types(), and newrow(). They process the column names, column types, and row data, respectively. Each call to VM::step() updates all of the column, type, and row information.

The use of VM is illustrated in the query() function of the example, which is listed in Listing 8-18.

Listing 8-18. *The query() Function*

```
public static void query(SQLite.Database db)
  throws SQLite.Exception
{
  System.out.println("\nQuery Processing:\n");

  Row row = new Row();
  db.set_authorizer(new AuthorizeFilter());

  Vm vm = db.compile( "select * from foods limit 5;" +
                      "delete from foods where id = 5;" +
                      "insert into foods (type_id, name) values (5, 'Java');" +
                      "select * from foods limit 5" );
  do
  {
```

```
    while (vm.step(row))
    {
      System.err.println(row.print());
    }
  }
  while (vm.compile());
}
```

The `SQLite.Database::exec()` performs self-contained queries and has the following form:

```
void exec(String sql, Callback cb, String[] params)
```

The `params` array corresponds to `%q` or `%Q` parameters in the SQL statement. An example is shown in Listing 8-19.

Listing 8-19. *The `exec_query()` Function*

```
public static void exec_query(SQLite.Database db)
  throws SQLite.Exception
{
  System.out.println("\nExec Query:\n");

  String sql = "insert into foods (type_id, name) values (5, '%q')";
  ResultSet result = new ResultSet();

  String params[] = {"Java"};
  db.exec(sql, result, params);

  System.out.println("Result: last_insert_id(): " + db.last_insert_rowid());
  System.out.println("Result:        changes(): " + db.changes());
}
```

Note that this is not the same thing as parameter binding. Rather, this is `sprintf` style substitution using `sqlite3_vmprintf()` in the SQLite C API.

User-Defined Functions and Aggregates

The `SQLite.Function` interface is used to implement both user-defined functions and user-defined aggregates. The `hello_newman()` function in Java is illustrated in Listing 8-20.

Listing 8-20. *`hello_newman()` in Java*

```
class HelloNewman implements SQLite.Function
{
  public void function(FunctionContext fc, String args[])
  {
    if (args.length > 0)
    {
      fc.set_result("Hello " + StringUtil.join(args, ", "));
    }
```


Download from Wow! eBook <www.wowebook.com>

```
    else
    {
      fc.set_result("Hello Jerry");
    }
  }

  public void step(FunctionContext fc, String args[]){}
  public void last_step(FunctionContext fc)
  {
    fc.set_result(0);
  }

}
```

Notice that the `step()` and `last_step()` functions, while specific to aggregates, must also be implemented even though they do nothing in user-defined functions. This is because the `SQLite.Function` interface defines methods for both functions and aggregates. This class is registered using `SQLite.Database.create_function()`. The function's return type must also be registered using `SQLite.Database.function_type()`. Listing 8-21 illustrates using the Java implementation of `hello_newman()`.

Listing 8-21. *The* `hello_newman()` *Test Code*

```
// Register function
db.create_function("hello_newman", -1, new HelloNewman());

// Set return type
db.function_type("hello_newman", Constants.SQLITE_TEXT);

// Test
PrintResult r = new PrintResult();
db.exec("select hello_newman()", r);
db.exec("select hello_newman('Elaine', 'Jerry')", r);
db.exec("select hello_newman('Elaine', 'Jerry', 'George')", r);
```

JDBC

The Java extension also includes support for JDBC. To use the driver, specify `SQLite.JDBCDriver` as the JDBC driver's class name. Also, make sure you have `sqlite.jar` in your class path and the native library in your Java library path. The JDBC URLs to connect to a SQLite database have the format `jdbc:sqlite:/path`, where `path` has to be specified as the path name to the SQLite database, for example.

```
<A NAME="50520101_jdbc_driver">jdbc:sqlite://dirA/dirB/dbfile
jdbc:sqlite:/DRIVE:/dirA/dirB/dbfile
jdbc:sqlite:///COMPUTERNAME/shareA/dirB/dbfile
```

Currently, the supported data types on SQLite tables are `java.lang.String`, `short`, `int`, `float`, and `double`. Some support exists for `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`. A connection property `daterepr` is used to define behavior for these data types. A value of `daterepr=Julian` sets SQLite to interpret all `insert/updates` of these data types as a floating-point conversion to the equivalent Julian

Day. Any other value switches behavior to use the string formats `YYYY-mm-dd` for date, `HH:MM:SS` for time, and `YYYY-mm-dd HH:MM:SS.f` for timestamp.

Other data type mapping depends mostly on the availability of the SQLite pragmas `show_datatypes` and `table_info`. Enough basic database metadata methods are implemented such that it is possible to access SQLite databases with JDK 1.4 or newer and the iSQL-Viewer tool.

Listing 8-22 is a simple example (located in `SQLiteJDBCExample.java`) of using the JDBC driver to query the `foods` table.

Listing 8-22. *The SQLite JDBC Test Program*

```
import java.sql.*;
import SQLite.JDBCDriver;

public class SQLiteJDBCExample {

public static void main ( String [ ] args )
{
  try
  {
    Class.forName("SQLite.JDBCDriver");
    Connection c = DriverManager.getConnection( "jdbc:sqlite://tmp/foods.db",
                                                ""  // username (NA),
                                                ""  // password (NA));

    Statement s = c.createStatement();
    ResultSet rs = s.executeQuery ("select * from foods limit 10");
    int cols = (rs.getMetaData()).getColumnCount();

    while (rs.next())
    {
      String fields[] = new String[cols];

      for(int i=0; i<cols; i++)
      {
        fields[i] = rs.getString(i+1);
      }

      System.out.println("[" + join(fields, ", ") + "]");
    }
  }
  catch( Exception x )
  {
    x.printStackTrace();
  }
}
```

```
static String join( String[] array, String delim )
{
  StringBuffer sb = join(array, delim, new StringBuffer());
  return sb.toString();
}

static StringBuffer join( String[] array, String delim, StringBuffer sb )
{
  for ( int i=0; i<array.length; i++ )
  {
    if (i!=0) sb.append(delim);
    sb.append("'" + array[i] + "'");
  }

  return sb;
}

}
```

This example produces the following output:

```
['1', '1', 'Bagels']
['2', '1', 'Bagels, raisin']
['3', '1', 'Bavarian Cream Pie']
['4', '1', 'Bear Claws']
['6', '1', 'Bread (with nuts)']
['7', '1', 'Butterfingers']
['8', '1', 'Carrot Cake']
['9', '1', 'Chips Ahoy Cookies']
['10', '1', 'Chocolate Bobka']
['11', '1', 'Chocolate Eclairs']
```

Tcl

SQLite's author wrote and maintains the Tcl extension. All of SQLite's testing code is implemented in Tcl and thus uses the SQLite Tcl extension. It is safe to say that this extension is both stable and well tested itself. The Tcl extension is included as part of the SQLite source distribution. You can find complete documentation for this extension on the SQLite web site: `www.sqlite.org/tclsqlite.html`.

Installation

The SQLite GNU Autoconf script will automatically search for Tcl and build the Tcl extension if it finds it. The recommended way to build the TCL interface is to use one of the tarballs or precompiled bindings for the Tcl extension for either Linux and Windows available on the SQLite web site. A simple "configure && make" invocation should see the TCL extension compiled and installed in the default location, so TCL and TCL/Tk scripts can load the extension normally. You'll also find the requisite libraries in the

package repositories of various distributions. For example, under Debian or Ubuntu, you can add the Tcl bindings as follows:

```
fuzzy@linux $ sudo apt-get install libsqlite3-tcl
```

As this chapter is being written, there is also talk of including SQLite by default in TCL version 8.6, which will make the process even easier!

Connecting

The SQLite extension is located in the `sqlite3` package, which must be loaded using the `package require` directive. To connect to a database, use the `sqlite3` command to create a database handle. This command takes two arguments: the first is the name of the database handle to be created, and the second is the path to the database file. The following example illustrates connecting to a database:

```
#!/usr/bin/env tclsh

package require sqlite3

puts "\nConnecting."
sqlite3 db ./foods.db
```

The usual database connection rules apply. Passing the value `:memory:` instead of a file name will create an in-memory database. Passing in the name of a new file will create a new database, and so on. The database handle returned corresponds to a connection to the specified database; however, it is not yet open—it does not open the connection until you try to use it. The database handle is the sole object through which you work with the database.

To disconnect, use the `close` method of the database handle. This will automatically roll back any pending transactions.

Query Processing

The extension executes queries using the `eval` method, which can process one or more queries at a time. `eval` can be used in several ways. The first way is to iterate through all records in a script following the SQL code. The script will be executed once for each row returned in the result set. The fields for each row are set as local variables within the script. Here's an example:

```
puts "\nSelecting 5 records."
db eval {select * from foods limit 5} {
    puts "$id $name"
}
```

There is another form in which you can assign the field values to an array. To do this, specify the array name after the SQL and before the script. Here's an example:

```
puts "\nSelecting 5 records."
db eval {select * from foods limit 5} values {
    puts "$values(id) $values(name)"
}
```

Both of the previous code snippets produce the following output:

```
Selecting 5 records.

1 Bagels
2 Bagels, raisin
3 Bavarian Cream Pie
4 Bear Claws
5 Black and White cookies
```

If you don't provide a script, `eval` will return the result set. The result set is returned as one long list of values, leaving you to determine the record boundaries. You can see this behavior in the following statement:

```
set x [db eval {select * from foods limit 3}]
```

This will return a list in variable `$x` that is six elements long:

```
{1 1 Bagels 2 1 {Bagles, raisin} 3 1 {Bavarian Cream Pie}}
```

This corresponds to three records, each of which has three fields (`id`, `type_id`, and `name`).

For non-`select` statements, `eval` returns information regarding modified records, as illustrated in Listing 8-23.

Listing 8-23. *Examining Changes in Tcl*

```
db eval begin

puts "\nUpdating all rows."
db eval { update foods set type_id=0 }
puts "Changes             : [db changes]"

puts "\nDeleting all rows."
# Delete all rows
db eval { delete FROM foods }

puts "\nInserting a row."
# Insert a row
db eval { insert into foods (type_id, name) values (9, 'Junior Mints') }

puts "Changes             : [db changes]"
puts "last_insert_rowid() : [db last_insert_rowid]"

puts "\nRolling back transaction."
db eval rollback
```

The code in Listing 8-23 produces the following output:

```
Updating all rows.
Changes             : 415

Deleting all rows.

Inserting a row.
Changes             : 1
last_insert_rowid() : 1

Rolling back transaction.
Total records       : 415
```

Transaction scope can be handled automatically within Tcl code using the `transaction` method. If all code inside the `transaction` method's script runs without error, the `transaction` method will commit; otherwise, it will invoke a rollback. For example, if you wanted to perform the code from Listing 8-23 in a single transaction, you would have to check the status of each command after its execution. If it failed, then you would roll back the transaction and abort any further commands. The more commands you have to run in the transaction, the messier the code will get. However, all of this can be done automatically with the `transaction` method, as illustrated in Listing 8-24.

Listing 8-24. *Transaction Scope in Tcl*

```
db transaction {
puts "\nUpdating all rows."
db eval { update foods set type_id=0 }
puts "Changes             : [db changes]"

puts "\nDeleting all rows."

# Delete all rows
db eval { delete from foods }

puts "\nInserting a row."

# Insert a row
db eval { insert into foods (type_id, name) values (9, 'Junior Mints') }

puts "Changes             : [db changes]"
puts "last_insert_rowid() : [db last_insert_rowid]"
}
```

Now, if any of the commands fail, `transaction` will roll back all commands without having to check any return codes. T`ransaction` also works with your existing transactions, working within a transaction that's already started. It will work within the already-started transaction and not attempt a commit or rollback. If an error occurs, it just aborts the script, returning the appropriate error code.

User-Defined Functions

User-defined functions are created using the `function` method, which takes the name of the function and a Tcl method that implements the function. Listing 8-25 illustrates an implementation of `hello_newman()` in Tcl.

Listing 8-25. *`hello_newman()` in Tcl*

```
proc hello_newman {args} {
    set l [llength $args]
    if {$l == 0} {
        return "Hello Jerry"
    } else {
        return "Hello [join $args {, } ]"
    }
}

db function hello_newman hello_newman
puts [db onecolumn {select hello_newman()}]
puts [db onecolumn {select hello_newman('Elaine')}]
puts [db onecolumn {select hello_newman('Elaine', 'Jerry')}]
puts [db onecolumn {select hello_newman('Elaine', 'Jerry', 'George')}]
```

The code for listing 8-25 produces the following output:

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

PHP

Since the advent of PHP version 5, SQLite has been part of the PHP standard library. If you have PHP 5 or newer, you have SQLite installed on your system as well. Like many of the other languages we've discussed, PHP presents you with a wealth of interface options.

PHP 5.1 introduced a new database abstraction layer called PHP Data Objects (PDO), which has grown to become one of the most popular database abstraction layers for PHP. This API uses drivers to support a standard database interface. The PDO interface is an OO interface that is very similar to the OO interface in the SQLite extension. Since PDO is an abstraction layer, it is meant to accommodate many different databases and is therefore somewhat generic. Despite this, it is still possible for PDO drivers to provide access to database-specific features as well. As a result, the PDO drivers provide a complete OO interface that works well with all the features of SQLite.

Installation

Installation of PHP is quite straightforward but beyond the scope of this book. You can find detailed instructions on building and installing PHP in the documentation on the PHP web site: `www.php.net`.

Connections

There are two important issues to consider before you open a database: location and permissions. By default, the file path in PHP is relative to the directory in which the script is run (unless you provide a full path, relative to the root file system). So if you specify just a database name, you are opening or creating a database within the public area of your web site, and the security of that database file depends on how the web server is configured.

Since SQLite databases are normal operating system files just like HTML documents or images, it may be possible for someone to fetch them just like a regular document. This could be a potential security problem, depending on the sensitivity of your data. As a general security precaution, it is good idea to keep database files outside of public folders so that only PHP scripts can access them. As an alternative, more sophisticated security can be achieved by limiting access to the SQLite database file using `.htaccess` controls or equivalent security techniques.

Ultimately, whatever permissions are used must allow the web server process running PHP both read and write access to the database files. Both of these are administrative details that must be addressed on a case-by-case basis.

In PDO, connections are encapsulated in the `PDO` class. The constructor takes a single argument called a data source name (DSN). The DSN is a colon-delimited string composed of two parameters. The first parameter is the driver name, which is normally `sqlite` and corresponds to SQLite version 3. The second parameter is the path to the database file. If the connection attempt fails for any reason, the constructor will throw a PDOException. The following example connects to our SQLite `foods.db` database:

```
<?php
try {
  $dbh = new PDO("sqlite:foods.db");
} catch (PDOException $e) {
  echo 'Connection failed: ' . $e->getMessage();
}?>
```

Queries

PDO is made up of two basic classes, `PDO` and `PDOStatement`. `PDO` represents a connection, which internally contains a `sqlite3` structure, and `PDOStatement` represents a statement handle, which internally contains a `sqlite3_stmt` structure. The query methods in the PDO class closely follow the methods in the SQLite C API. There are three ways to execute queries:

- `exec()`: Executes queries that don't return any data. Returns the number of affected rows, or `FALSE` if there is an error. This mirrors `sqlite3_exec()`.
- `query()`: Executes a query and returns a `PDOStatement` object representing the result set, or `FALSE` if there is an error.

- `prepare()`: Compiles a query and returns a `PDOStatement` object or `FALSE` if there is an error. This offers better performance than `query()` for statements that need to be executed multiple times, because it can be reset, avoiding the need to recompile the query.

Additionally, transaction management in PDO can be performed through a method invocation using `beginTransaction()`, `commit()`, and `rollback()`. Listing 8-26 shows a basic example of using the PDO class to open a database and perform basic queries within a transaction.

Listing 8-26. *Basic Queries with PDO*

```
<?php

try {
  $dbh = new PDO("sqlite:foods.db");
} catch (PDOException $e) {
  echo 'Connection failed: ' . $e->getMessage();
}

$dbh->beginTransaction();
$sql = 'select * from foods limit 10';
foreach ($dbh->query($sql) as $row) {
  print $row['type_id'] . " ";
  print $row['name']    . "<br>";
}

$dressing = $dbh->quote("Newman's Own Vinegarette");
$dbh->exec("insert into foods values (NULL, 4, $dressing)");
echo $dbh->lastInsertId();
$dbh->rollback();
?>
```

PDO uses the `setAttribute()` and `getAttribute()` methods to set various database connection parameters. The only parameter that applies to SQLite is `PDO_ATTR_TIMEOUT`, which sets the busy timeout.

The `prepare()` method uses a SQLite statement to navigate through a result set. It also supports both named and positional parameters. To bind a SQL parameter, there must be some associated variable in PHP to bind it to. PHP variables are bound to parameters using `PDOStatement::bindParam()`, which has the following declaration:

```
nool bindParam ( mixed parameter, mixed &variable,
                 int data_type, int length,
                 mixed driver_options );
```

The `parameter` argument specifies the SQL parameter. For numbered parameters, this is an integer value. For named parameters, it is a string. The `variable` argument is a reference to a PHP variable to bind. The `data_type` argument specifies the data type of the variable. Finally, the `length` argument specifies the length of the value if the variable is a string. The value specifies the maximum length of the string to return. Listing 8-27 is a complete example of using positional parameters.

Listing 8-27. *Positional Parameters with PDO*

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'select * from foods where type_id=? And name=?';
$stmt = $dbh->prepare($sql);
$type_id = 9;
$name = 'JujyFruit';
$stmt->bindParam(1, $type_id, PDO_PARAM_INT);
$stmt->bindParam(1, $name, PDO_PARAM_STR, 50);
$stmt->execute();
?>
```

Named parameters work in a similar fashion. When you bind these parameters, you identify them by their names (rather than integers). Listing 8-28 is a modification of the previous example using named parameters.

Listing 8-28. *Named Parameters with PDO*

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'select * from foods where type_id=:type And name=:name;';
$stmt = $dbh->prepare($sql);
$type_id = 9;
$name = 'JujyFruit';
$stmt->bindParam('type', $type_id, PDO_PARAM_INT);
$stmt->bindParam('name', $name, PDO_PARAM_STR, 50);
$stmt->execute();
?>
```

Download from Wow! eBook <www.wowebook.com>

PDO also allows you to bind columns of result sets to PHP variables. This is done using `PDOStatement::bindColumn()`, which has the following declaration:

```
bool bindColumn (mixed column, mixed &param, int type)
```

The column argument refers to either the name of the column or its index in the SELECT clause. The `param` argument is a reference to a PHP variable, and the `type` argument specifies the type of the PHP variable. The following example binds two variables `$name` and `$type_id` to a result set:

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'select * from foods limit 10';
$stmt = $dbh->prepare($sql);
$stmt->execute();
$name;
$type_id;
$stmt->bindColumn('type_id', $type_id, PDO_PARAM_INT);
$stmt->bindColumn('name', $name, PDO_PARAM_STR);
```

```
while ($row = $stmt->fetch()) {
  print "$type_id $name <br>";
}
```

User-Defined Functions and Aggregates

User-defined functions are implemented using `sqliteCreateFunction()`. Aggregates are implemented using `sqliteCreateAggregate()`. `sqliteCreateFunction()` has the following form:

```
void PDO::createFunction ( string function_name,
                           callback callback,
                           int num_args );
```

The arguments are defined as follows:

- `function_name`: The name of the function as it is to appear in SQL
- `callback`: The PHP (callback) function to be invoked when the SQL function is called
- `num_args`: The number of arguments the function takes

The following example is a PHP implementation of `hello_newman()`:

```
<?php
function hello_newman() {
    return 'Hello Jerry';
}

$db = new PDO("sqlite:foods.db");
$db->createFunction('hello_newman', hello_newman, 0);
$row = $db->query('SELECT hello_newman()')->fetch();
print $row[0]
?>
```

You create aggregates in a similar fashion using the `sqliteCreateAggregate` function, declared as follows:

```
void PDO::createAggregate ( string function_name,
                            callback step_func,
                            callback finalize_func,
                            int num_args )
```

The following code is a simple implementation of the previous `SUM` aggregate, called `phpsum`:

```
<?php
function phpsum_step(&$context, $value) {
    $context = $context + $value;
}
```

```
function phpsum_finalize(&$context) {
    return $context;
}

$db = new PDO("sqlite:foods.db");
$db->createAggregate('phpsum', phpsum_step, phpsum_finalize);
$row = $db->query('select phpsum(id) from food_types')->fetch();
print $row[0]
?>
```

Summary

You've now experienced a brief survey of several different language extensions and how they work with SQLite. Although using SQLite with the C API is quite straightforward, using SQLite in language extensions is considerably easier. Many of the concepts are very similar, and many of the extensions map in a one-to-one fashion to the underlying SQLite C API. As you can see, there are many things in common even in cases where an extension conforms to a language-specific database API. All queries ultimately involve a connection object of some kind, which maps to an internal `sqlite3` structure, and a statement or cursor object, which internally maps to a `sqlite3_stmt` structure.

These extensions make using SQLite convenient and easy, increasing accessibility to many more applications ranging from system administration to web site development. There has been an explosion in the number of extensions in recent years, and you can find out about the (literally) hundreds of available extensions on the SQLite wiki at `www.sqlite.org/cvstrac/wiki?p=SqliteWrappers`.

CHAPTER 9

iOS Development with SQLite

You might be a seasoned SQLite user or developer, or perhaps you're reading this book as your first foray into the SQLite world. Regardless of your background, you've almost certainly been using SQLite every day without even realizing it. That's because SQLite is one of the most popular database facilities used in a huge range of mobile devices, from iPhones and iPods to the latest tablets, iPads, media players, and more.

In this chapter, we'll take a slightly different angle on developing with SQLite. We'll focus our coverage on Apple's implementation and support for SQLite across its entire fleet of mobile devices. Where previously we might have referred to this as "iPhone development," Apple has unified the naming for its operating system on mobile devices under the name iOS.

Yet another chapter on showing how one API maps to another would likely leave you feeling a little underwhelmed, because by now you're probably able to pick up many aspects of SQLite wrappers quickly and easily. We will instead provide both coverage of the iOS SQLite technology stack and also focus our examples on building a real, working iOS SQLite-based application. You will be the envy of trivia night and quiz show aficionados when you arrive sporting your very own personally coded *Seinfeld* food trivia iOS app! We'll call it *iSeinfeld*…what else?

We'll also cover some of the developer considerations required for successfully maintaining iOS applications on a device over time.

Prerequisites for SQLite iOS Development

To do any development work with SQLite for iOS, you'll need to equip yourself with the necessary tools. These tools aren't strictly speaking solely for the benefit of SQLite development: once completed, you'll have a full iOS development environment for building both SQLite-focused and other Objective-C applications for iPhones, iPods, and iPads.

The three broad steps you need to complete are as follows:

1. Sign up for the Apple Developer program.
2. Download and install Xcode and the Apple IDE.
3. Download and install the iOS SDK.

We'll cover each of these *very* briefly so that the SQLite work can begin!

Signing Up for Apple Developer

To develop and publish applications for the iPad, iPhone, or any other future "i" devices, you need to sign up to Apple's developer program. Ordinarily, we'd just mention this in passing and leave you to your own devices to get signed up. However, it's worth pointing out one nuance of the sign-up process that might, at first glance, appear to dissuade many budding iOS developers.

Depending on which page you read on the Apple Developer web site, it can appear that you need to sign up for a $99 iOS Developer Program membership. Notably, this is displayed prominently on the iOS Developer home page, `http://developer.apple.com/programs/ios/`. Although it is true that you need to buy such a membership to eventually *distribute* applications via the iTunes store (and jump the various hurdles of application approval), you only need to register as an Apple developer in order to gain access to the tools, SDKs, and so on, for iOS development.

Simple registration as a developer is free! It can be a little confusing when following the links on the Apple site, but at the time of writing this book, if you head straight to `http://developer.apple.com/programs/register/`, you'll have no problems signing up.

Naturally, if you come to love building SQLite-based iOS apps, you can upgrade your registration at any time to qualify for distributing your applications via iTunes.

Downloading and Installing Xcode and the iOS SDK

Once you've signed up as an Apple developer, you need to download Apple's integrated development environment, known as Xcode, and the iOS SDK. In another of Apple's quirks, the iOS SDK *includes* Xcode. That means you don't need to download Xcode and a separate SDK. The iOS SDK includes bundled versions of SQLite, which we'll discuss shortly, so you also can avoid downloading and maintaining separate versions of SQLite libraries if you choose. However, this bundled approach can also can trick first-timers to Apple's iOS development world, because one's instinct is to download Xcode as the IDE you plan to use and then add in the iOS SDK. You'll find yourself needlessly downloading a few extra gigabytes of software if you take this approach.

Apple makes downloading the iOS SDK (with the included Xcode) straightforward. Once you log in with your developer credentials, follow the download link on the developer home page, and choose the option Xcode 3.2.4 and iOS SDK 4.1. This should download a file named `xcode_3.2.4_and_ios_sdk_4.1.dmg`, which is approximately 3.3GB in size.

■ **Note** The current 3.2.*x* releases of Xcode and 4.*x* releases of the iOS SDK will only work on Apple machines running Snow Leopard or newer releases of OS X. For those of you still running Leopard (10.5.*x*) or older, Apple does make available earlier compatible versions of Xcode and iPhone SDKs (as they were called). Searching for these can be a hit-and-miss affair. If you have difficulty finding them on the Apple site, we suggest a Google search of the Apple forums, where avid developers keep track of where Apple moves the older `*.dmg` files for earlier SDK releases.

With the iOS SDK downloaded, you can now proceed to installation. Open the `.dmg` file, and you will be presented with three items, as shown in Figure 9-1. These are an informative "About" file, the SDK installer itself, and a packages directory.



Figure 9-1. *Your iOS SDK package contents*

Run the iOS SDK installer now (or iPhone SDK installer, depending on the version you used). After a few moments of disk activity, you should see the installer's welcome page, shown in Figure 9-2.



Figure 9-2. *iOS SDK package installation welcome*

With one exception, there's no need to walk you through the various installation screens. Go ahead and choose the defaults, or add additional options as you see fit throughout the installation. However, it's worth noting some of the options when you reach the Installation Type screen, shown in Figure 9-3.



Figure 9-3. *iOS SDK installation types*

Here, the iOS SDK installer is prompting for historic versions of the SDK to include in the installation. As well as always including the most current version of the SDK, you have the choice of targeting older versions. Why would you want to do this? For the purposes of our SQLite development throughout the rest of the chapter, the defaults are fine. But if you might be doing future development where you need to target older iPhones and iPads, you might want to include the older SDKs. Although this is probably a rare occurrence, people developing for corporate or enterprise environments where the latest iPads are still a future promise might well find they need to target the original iPhone to meet their users' hardware specifications.

Continue from here to the end of the installation, and you should see a satisfying and quite large success message, as shown in Figure 9-4.



Figure 9-4. *A green check mark shows a successful iOS SDK installation.*

With your installation complete, you are ready to code your first iOS SQLite-based application.

Alternatives to Xcode

For those brave, experienced, or both, there are alternatives to Xcode for iOS development in general and SQLite-based iOS development in particular. Feel free to use any or all of these if you're already comfortable with iPhone development and prefer them to Xcode.

PhoneGap: An open source development framework for building cross-platform mobile applications. `www.phonegap.com`

Appcelerator Titanium: Another open source cross-platform framework for mobile applications. `www.appcelerator.com`

Each of these has excellent support for SQLite development for iOS and other platforms. Note that Apple have just relaxed their notoriously strict and some would say capricious application rules that penalized coding in different languages or environments and cross-compiling or translating to target Objective-C and iOS. So, you have some more leeway to explore these tools and launch the next great SQLite-based iOS application.

Building the iSeinfeld iOS SQLite Application

With a working development environment now in place, you are ready to build your SQLite-based iOS application. Like all good developers, we'll need some idea of our application's requirements in order to properly design our application and know when it is finished. Here are our very succinct iSeinfeld requirements:

> Build an iOS application for iPhones and iPads that shows the foods mentioned in *Seinfeld* by order of popularity.
>
> For any popular food, show how often the food is mentioned over all shows of *Seinfeld*.
>
> Indicate the show in which a food first appeared.

That seems pretty simple. The code we develop will cover the core aspects of working with SQLite on iOS systems. We'll cover opening databases, or creating them if they don't already exist; manipulating data within the database; and the all-important UI elements of bringing your SQLite data to the attention of the user!

Here are the steps we will cover in this chapter. We'll try to keep the Objective-C and Cocoa touch–specific aspects to a minimum and highlight the all-important SQLite dimension of the work wherever possible.

1. Create a new project in Xcode.
2. Add the SQLite framework to the project.
3. Prepare our foods database for inclusion in the iOS application.
4. Define the classes that map our food data from SQLite to our code and that present that data on the screen for users to see.
5. Access and query the SQLite database.
6. Put on the final polish before publishing the application.

Let's get started.

Step 1: Creating a New Xcode Project

Launch Xcode to get started. If this is the first time you've started Xcode (or if you haven't changed the defaults), you'll see a Welcome to Xcode splash screen on top of the Xcode environment. Dismiss this for now—in fact, if you don't want to see this again, also uncheck the "show at launch" option.

With Xcode running, choose File ➤ New Project. For our iSeinfeld application, we want to use a navigation-based application as the project template. This is because we'll be using master and detail screens for various representations of our SQLite foods data. Figure 9-5 shows the new project options in Xcode.

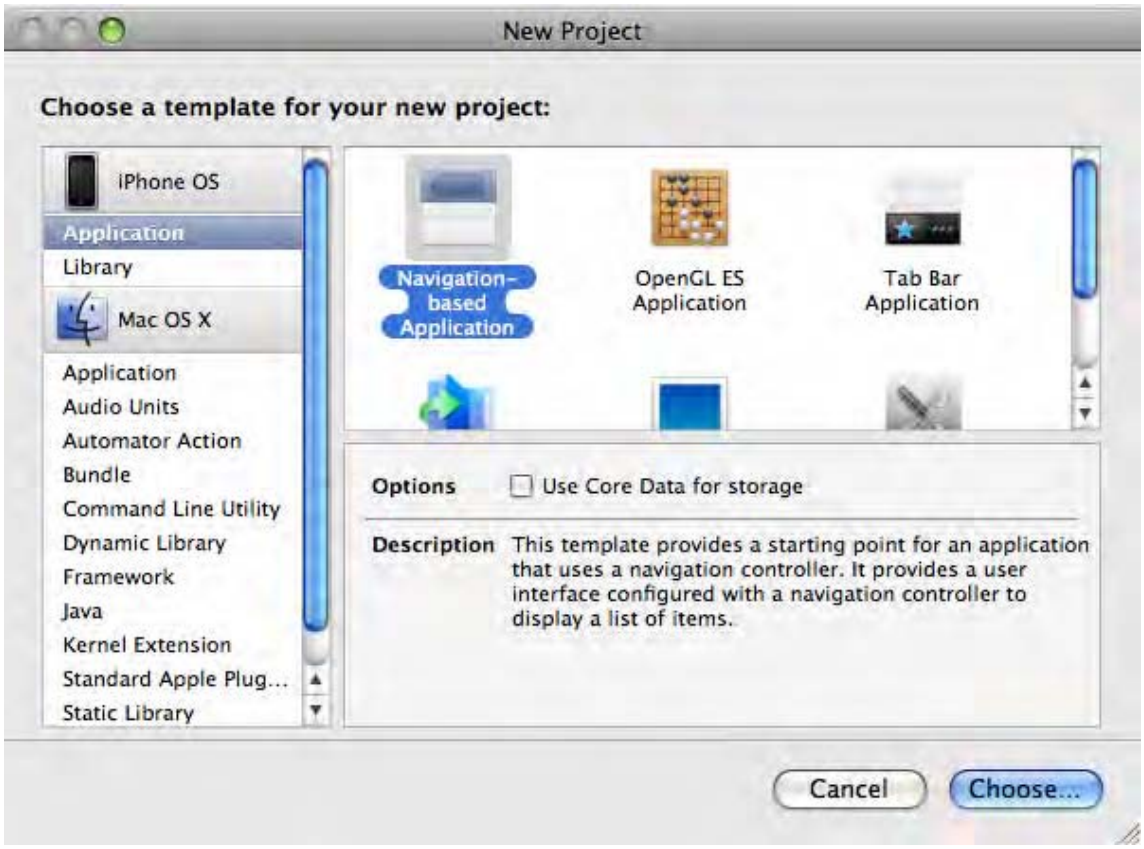


Figure 9-5. *Creating a new navigation-based application for our SQLite project*

Choose a meaningful name for your project. We've called ours iSeinfeld, but feel free to adopt any memorable or meaningful name. By default, this will be the name displayed for your application on the home screen of iOS devices (though this can be changed later). Your freshly created project should look similar to our project shown in Figure 9-6.

Download from Wow! eBook <www.wowebook.com>

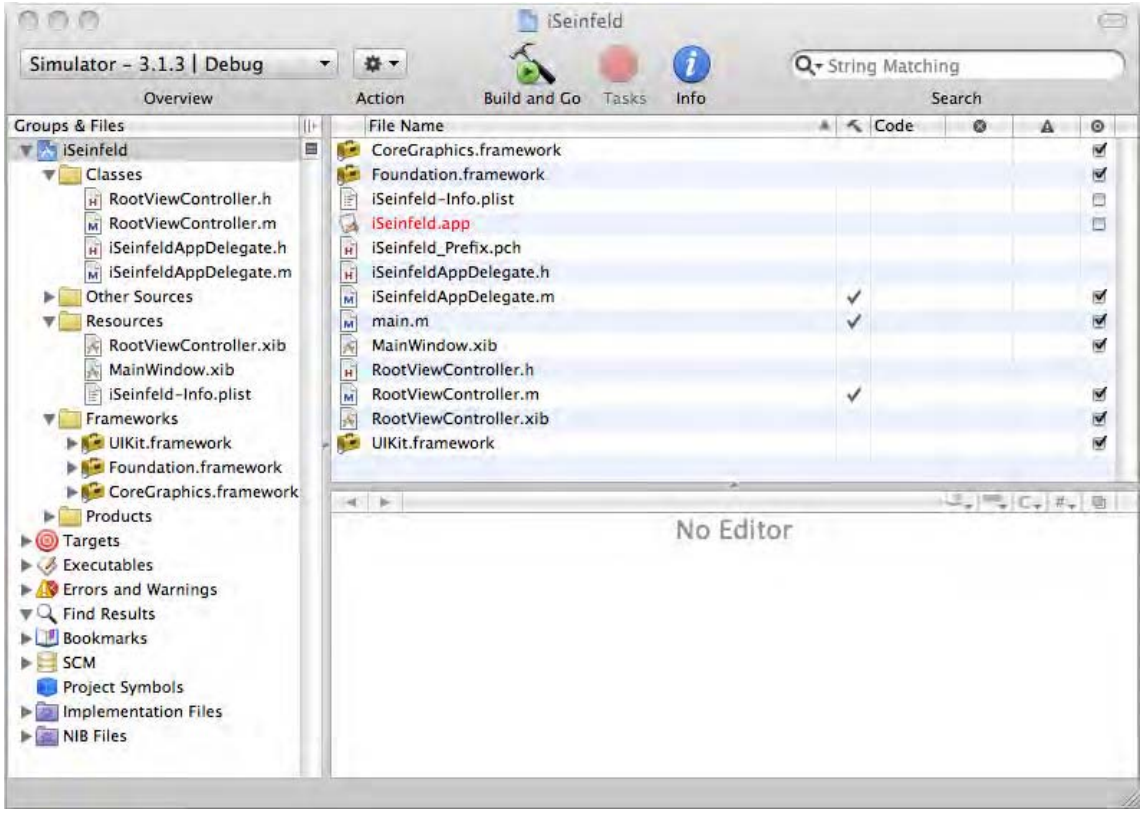


Figure 9-6. *The new iSeinfeld project in Xcode*

Step 2: Adding the SQLite Framework to Your Project

With your blank iSeinfeld project in place, we now need to add the SQLite library to the list of frameworks so our future code can benefit from, and use, the SQLite API.

From the left pane of your Xcode project, open the Frameworks folder. Ctrl+click and choose Add ➤ Existing Frameworks. In the dialog box, navigate to the directory `/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS`*`x.y.z`*`.sdk/usr/lib/`, where *`x.y.z`* is the version number of the iPhone firmware version your Xcode and iOS target. Depending on the options you choose at Xcode and iOS SDK installation time, you may see multiple versioned directories. Our recommendation is to choose the latest (highest) version.

■ **Note** If you're using the latest iOS version 4 releases, those directories may change to read `iOS` in place of `iPhone`. Don't worry, the underlying library is the same.

From that directory, choose the `libsqlite3.dylib` entry. You will likely also see a `libsqlite3.0.dylib` or similar file. The `libsqlite3.dylib` entry is a soft link to the latest version of the `dylib` file, in this case `libsqlite3.0.dylib`. We recommend always choosing the soft link to avoid unnecessarily specific version choices.

Choose this file, and you'll be presented with an options dialog box asking for reference type and text encoding. We don't need any of the alternative behavior offered here, so it's safe to just click Add to continue with the defaults. You should now be returned to your Xcode project. Under the Frameworks folder on the left pane, you should now see the `libsqlite3.dylib` entry, as shown in Figure 9-7.

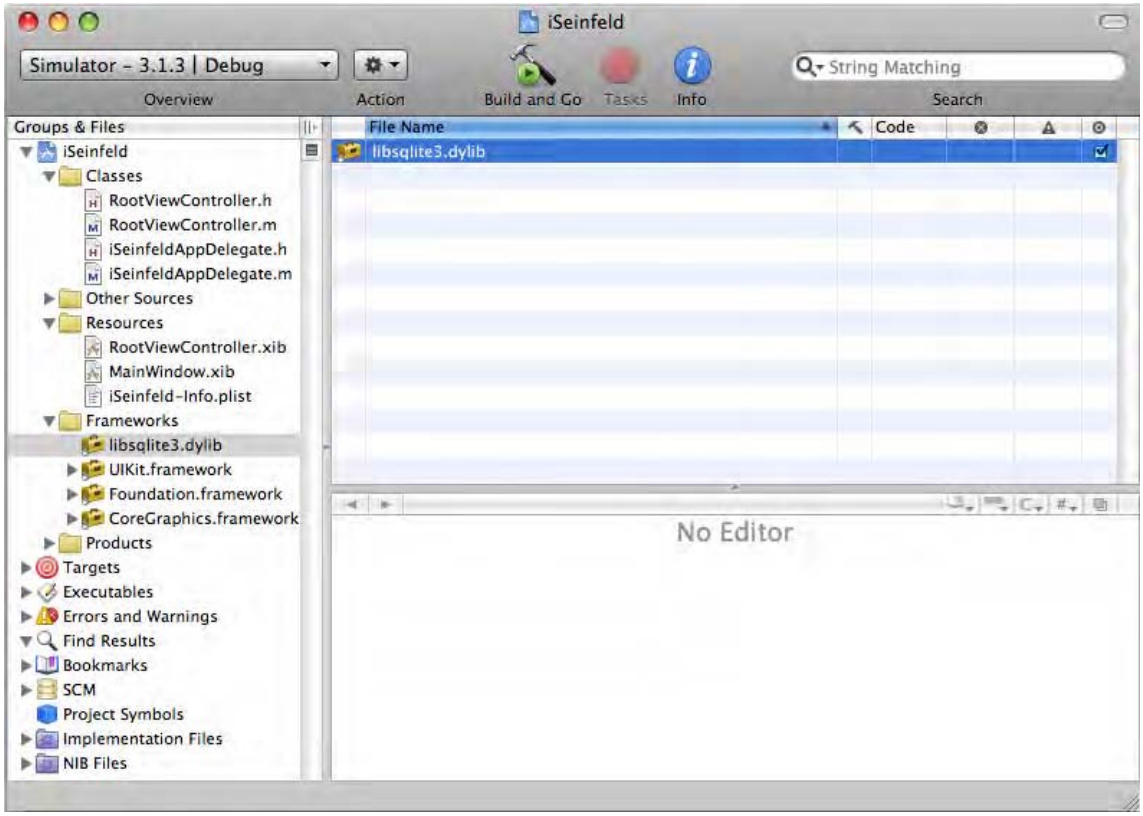


Figure 9-7. *The SQLite* `libsqlite3.dylib` *library added to the iSeinfeld project in Xcode*

Step 3: Preparing the Foods Database

We have two options for getting a working database in place for our project. We can step through the process of creating a new database, creating tables, and inserting data; or we can deploy an existing database for use with our iSeinfeld application.

Creating everything fresh by hand would see us walk through the SQLite API for `sqlite3_open_v2()`, `sqlite3_prepare()`, and so on, and countless iterations of `sqlite3_exec()` and so forth. Because of the very close similarities between C and Objective-C, this would basically be a repeat of Chapter 6. We think that would be a waste of the chapter. More importantly, when you build and release the next great iOS application yourself, you almost certainly will package it with an existing database (though you might well provide features to add new data and work with existing data). We'll adopt this second approach, but we'll also cover the options of creating a fresh database, as well as the typical SQL statements you'd expect for data manipulation within SQLite. We are going to add our `foods.db` SQLite database to our project so we can get working with our data faster!

If you're following the default directory layout for Xcode, you'll find your iSeinfeld SQLite project in the `/Users/`*`user_name`*`/Documents/iSeinfeld` directory (where *`user_name`* is your username). Copy your `foods.db` file to this directory in order to keep all of your project data, code, and resources together. On the left pane of your Xcode project, open the Resources folder. Ctrl+click, and choose Add ➤ Existing files. In the Add File dialog box, navigate to the `/Users/`*`user_name`*`/Documents/iSeinfeld` directory, and choose the `foods.db` file, as shown in Figure 9-8.

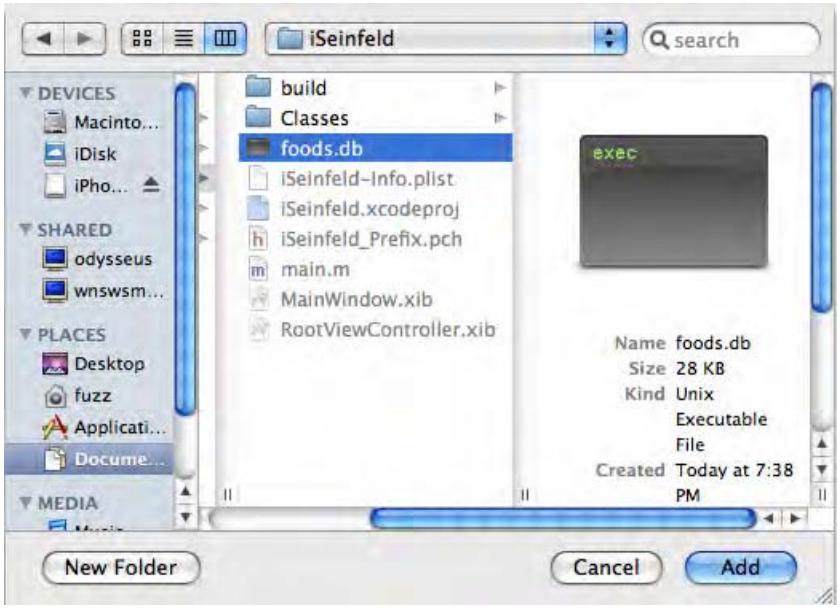


Figure 9-8. *Adding an existing SQLite database to your Xcode project*

Once again, Xcode will prompt with several options for this file's inclusion in the project. We only need the default behavior here, so click Add to have the `foods.db` file added to your project. Your Xcode resources folder should now list the `foods.db` SQLite database, as shown in Figure 9-9.

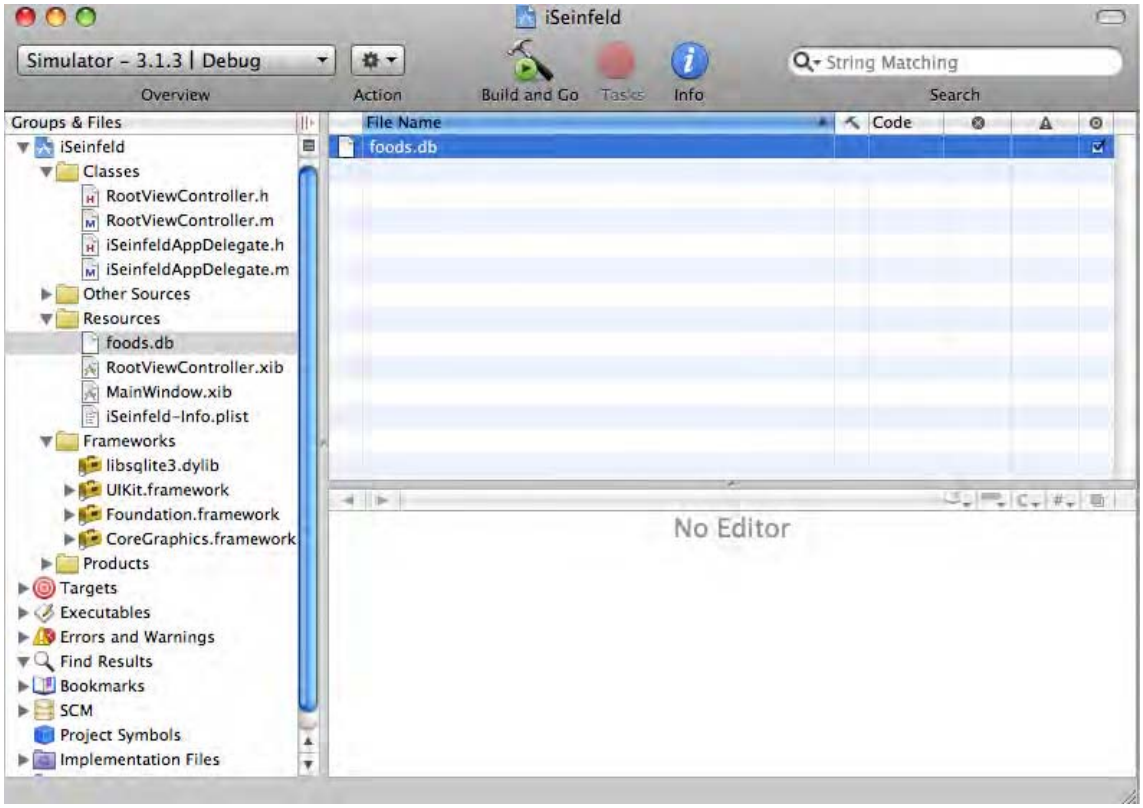


Figure 9-9. *The iSeinfeld project with foods SQLite database included as a resource*

Your iSeinfeld iOS application now has the SQLite libraries in place and a populated database included in the project. It's time to start coding!

Step 4: Creating Classes for the Food Data

In addition to the default classes created with our new project, we will need two classes to represent the data from our SQLite database as objects within the application. Well need a `Food` class to represent one kind of food and the data we want to use from foods within our application. We'll also need a view controller class to manage the interaction of our interface (the view) with the model that represents our foods.

IOS ANDMODEL-VIEW-CONTROLLER DEVELOPMENT

The Model-View-Controller approach to dividing up functionality in a program has become both popular and mature over recent years. The good news is the iOS approach to development, and Xcode tools, are designed to harmoniously work in an MVC world.

Our little example isn't really big enough to show off the natural MVC tool set and ways of working within Xcode, but you can rest assured that the support and approach are there when needed.

The Food Class

Within your iSeinfeld project in Xcode, Cmd+click the Classes folder, and choose Add ➤ New File. Call your new class `Food`. We want to derive Food from the base Objective-C class and use the `NSObject` subclass. We'll provide three data members: `Name`, `Popularity`, and `First Episode`. We'll also use a custom `init` function (think constructor for those familiar with other object-oriented languages) to create a `Food` instance more easily. Listing 9-1 shows the `Food.h` header.

Listing 9-1. *Food.h*

```
//  Food.h
//  iSeinfield
//
//  Created by Grant Allen on 9/9/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <Foundation/Foundation.h>


@interface Food : NSObject {
  NSString *first_episode;
  NSString *popularity;
  NSString *name;
}

@property (nonatomic, retain) NSString *first_episode;
@property (nonatomic, retain) NSString *popularity;
@property (nonatomic, retain) NSString *name;

-(id)initWithName:(NSString *)n first_episode:(NSString *)f popularity:(NSString *)t;

@end
```

At this point, there's no sign of our SQLite API calls, but you can probably already see where the `Food` class will fit in. Rows returned from our particular SQLite queries will be used as seed data in the `initWithName` function to create instances of the `Food` object.

With our header in place, we actually need an implementation. Open the `Food.m` file, as shown in Listing 9-2, and flesh out the `initWithName` function to assign the parameters `n`, `f`, and `t` to the instance variables.

Listing 9-2. *`Food.m`*

```
//  Food.m
//  iSeinfield
//
//  Created by Grant Allen on 9/9/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import "Food.h"

@implementation Food
@synthesize first_episode, popularity, name;

-(id)initWithName:(NSString *)n first_episode:(NSString *)f popularity:(NSString *)t {
  self.name = n;
  self.first_episode = f;
  self.popularity = t;

  return self;
}

@end
```

Pretty simple stuff, even if you've never coded in Objective-C before. For those of you new to Objective-C, you'll notice a `@synthesize` directive, followed by each of the member variables for our class. That's an Objective-C trick to automatically create getters and setters on your behalf, saving you from the effort of writing six dull methods to do the same thing.

The FoodViewController Class

Our mini-specification for the iSeinfeld application asked us to display detailed information for a given food held in our SQLite database. From our `Food` class, you can guess that the details will be `popularity` and `first_episode`. We need a new class to manage the view of these details (in the MVC sense) and act as the controller with the model. Enter the `FoodViewController` class. Create a new class as before, but this time choose the `UIViewController` subclass. Name it `FoodViewController`. We also will need a View XIB file to act as our canvas for the detail view. Under Resources, choose Add ➤ New File, and from the User Interfaces group choose View XIB. Call this XIB `FoodViewController.xib` to give you a mental reminder that this UI element is used with the `FoodViewController` `UIViewController` class.

Our `FoodViewController` class needs to define two outlets for the detail data to interface with the View XIB. Listing 9-3 shows the definition of `FoodViewController.h` and these required outlets.

Listing 9-3. *FoodViewController.h*

```
//  FoodsViewController.h
//  iSeinfield
//
//  Created by Grant Allen on 9/9/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <UIKit/UIKit.h>


@interface FoodViewController : UIViewController {
  IBOutlet UITextView *first_episode;
  IBOutlet UITextView *popularity;
}

@property (nonatomic, retain) IBOutlet UITextView *first_episode;
@property (nonatomic, retain) IBOutlet UITextView *popularity;

@end
```

We've chosen `UITextView` as the outlet modifier, because we know we'll be displaying textual data and numbers. The implementation of this class in `FoodViewController.m` contains a few pages of default methods and many commented-out extras. We won't waste paper showing those. Simply ensure that you import the header file, and in the `FoodViewController` implementation you synthesize the two member variables, like this:

```
#import "FoodViewController.h"

@implementation FoodViewController

@synthesize foodID, typeID;

// more template methods etc.
// …
```

Your choice for layout for `FoodViewController.xib` is entirely up to your artistic desire. As a guide, we've created a simple screen with two `Text Field` elements and two `Label` elements and wired them up in Interface Builder to the outlets defined earlier. Figure 9-10 shows our finished XIB layout.

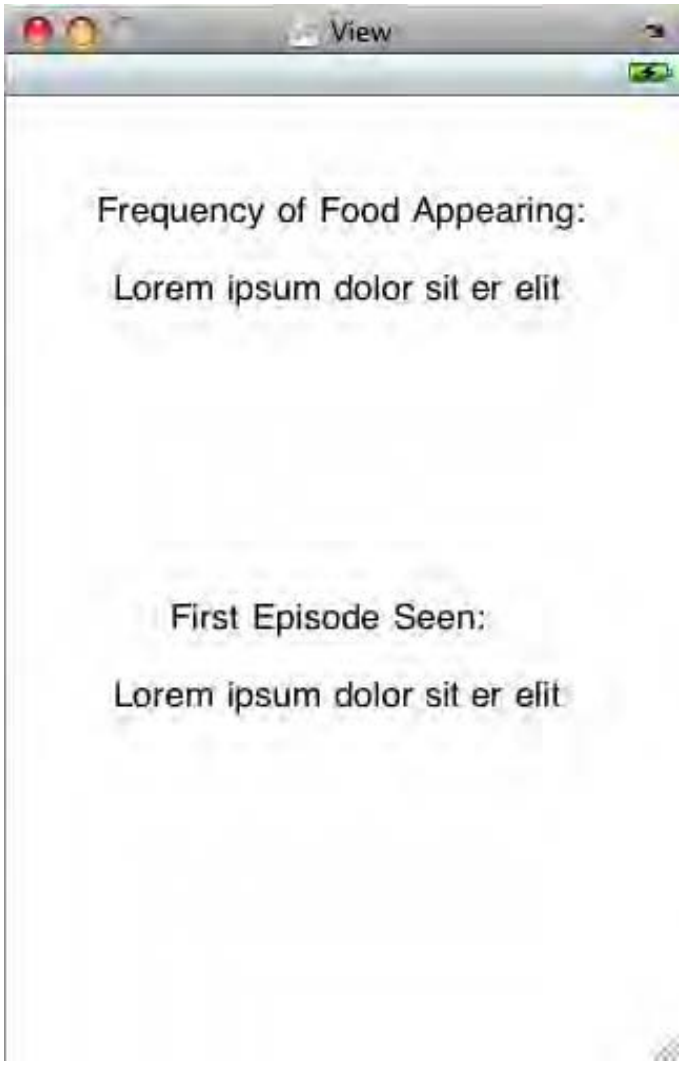


Figure 9-10. *The iSeinfeld food detail screen in Xcode Interface Builder*

Your iSeinfeld iOS application now has the ability to show data from SQLite, but we've yet to actually access the data and use our SQLite API. We can't keep up the suspense any longer. Let's get to the database!

Step 5: Accessing and Querying the SQLite DB

When our iSeinfeld application starts, we want it to find and open our `foods.db` database. At that point, we can then exercise all our SQLite C APIs to query, control, and otherwise use the database. To find and open the database at launch time, we want to extend the `iSeinfeldAppDelegate` class that was automatically created with our project. Listing 9-4 shows the changes we've made to the `iSeinfeldAppDelegate.h` file.

Listing 9-4. *`iSeinfeldAppDelegate.h` File*

```
//  iSeinfeldAppDelegate.h
//  iSeinfeld
//
//  Created by Grant Allen on 9/9/10.
//  Copyright __MyCompanyName__ 2010. All rights reserved.
//

@interface iSeinfeldAppDelegate : NSObject <UIApplicationDelegate> {

  UIWindow *window;
  UINavigationController *navigationController;

    // Details of your database
    NSString *databaseName;
    NSString *databasePath;

    // Our iSeinfeld food array, for storing our foods
    NSMutableArray *foods;

}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController *navigationController;
@property (nonatomic, retain) NSMutableArray *foods;

@end
```

We've outlined the member variables that will house the path and name of our SQLite database. We've also defined an `NSMutableArray` for holding our collection of foods. The interface definition is straightforward. The devil is in the detail.

We'll extend our applications behavior in the `iSeinfeldAppDelegate.m` file. Open the file, and you'll see a healthy number of automatically generated methods that have names such as `applicationWillTerminate` and so on. We want to inject behavior into our application once it has launched, so we'll modify the `applicationDidFinishLaunching` function.

```
// … lots of code above here
- (void)applicationDidFinishLaunching:(UIApplication *)application {

  // Set up our foods database as the databaseName
  databaseName = @"foods.db";

  // Get the path to the documents directory and append the database name
  NSArray *documentPaths = NSSearchPathForDirectoriesInDomains(
                           NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDir = [documentPaths objectAtIndex:0];
  databasePath = [documentsDir stringByAppendingPathComponent:databaseName];

  // Check for existing database and create if necessary
  [self checkAndCreateDatabase];

  // Query the database for all foods and populate the array
  [self readFoodsFromDatabase];

// …more code follows
```

We use several built-in features to find the `foods.db` SQLite file in the project. `NSSearchPathForDirectoriesInDomains`, `NSDocumentDirectory`, and `NSUserDomainMask` are all constants provided by the iOS SDK to find the paths within the running environment. In this case, these lines are effectively a shortcut to the current application's location, wherever that may be. We end up with the location of the `foods.db` file in the `databasePath` variable.

Our logic then splits into code that handles locating the database (or creating it afresh from the project) and code to query our database. The code for locating the database is not really SQLite specific, covering simple file-handling operations. See the `iSeinfeldAppDelegate.m` file included with the source code for how this works.

Opening the database and using our SQLite APIs to query the data is where the action is at. Our `readFoodsFromDatabase` function does this work, so let's review it in detail.

```
-(void) readFoodsFromDatabase {
  // Setup the database object
  sqlite3 *database;

  // Initialize the foods Array
  foods = [[NSMutableArray alloc] init];

  // Open the database
  if(sqlite3_open_v2([databasePath UTF8String], &database) == SQLITE_OK) {
    // Setup the SQL Statement and compile it for faster access
    const char *sqlStatement =
      "select
       count(foods_episodes.episode_id) popularity,
       min(foods_episodes.episode_id) first_seen,
       foods.name
       from foods inner join foods_episodes on foods.id = foods_episodes.food_id
       group by foods.name
       order by popularity desc, name
       limit 50 ";
```

Download from Wow! eBook <www.wowebook.com>

```
    sqlite3_stmt *compiledStatement;
    if(sqlite3_prepare_v2(database, sqlStatement, -1, &compiledStatement, NULL)
      == SQLITE_OK) {
      // Loop through the results and add them to the foods array
      while(sqlite3_step(compiledStatement) == SQLITE_ROW) {
        // Read the data from the result row
        NSString *aFreq = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 0)];
        NSString *aPop = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 1)];
        NSString *aName = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 2)];

        // Create a new food object with the data from the database
        Food *food = [[Food alloc] initWithName:aName
                        first_episode:aFreq popularity:aPop];

        // Add the food object to the foods Array
        [foods addObject:food];

        [food release];
      }
    }
    // finalize the statement
    sqlite3_finalize(compiledStatement);

  }
  sqlite3_close(database);

}
```

Working through that code, you can see some familiar API calls. We start with defining a sqlite3 object named `database`:

```
  sqlite3 *database;
```

After defining an array for our foods, we the try to open our database using the `sqlite3_open_v2()` API call.

```
if(sqlite3_open_v2([databasePath UTF8String], &database) == SQLITE_OK)
```

The `if` construct should be familiar to you regardless of your preferred programming language. And we're looking for the return value `SQLITE_OK` from the `sqlite3_open_v2()` call. This example could be straight from Chapter 5, 6, or 7. But note the odd handling of the `databaseName` variable, which we'd previously defined in `iSeinfeldAppDelegate` to contain our database's full path and file. Because Objective-C uses `NSString` as the base string class but SQLite's C API uses standard C-style strings/chars, we need to cast the `NSString` `databaseName` to a C string so that the SQLite API understands the database name that it should open. The built-in UTF8String option performs this conversion for us.

With our database open, we then create a `sqlite3_stmt` object called `compiledStatement`. This is assigned our query text.

```
    sqlite3_stmt *compiledStatement;
```

Again, that's eerily like straight C syntax: the differences with Objective-C are not that large once you start working with it. We use `sqlite3_prepare_v2` to prepare the statement and `sqlite3_step` to step through the results.

As we step through the results, we retrieve the column data with statements of the following form (note this is one line of code, split here for clarity):

```
NSString *aName = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 2)];
```

You'll notice an old friend here. The `sqlite3_column_text()` API call is used here to retrieve the text of the "name" column in our results. We can use any of the `sqlite3_column_xxx()` functions as appropriate. We then perform a conversion from the base C `char*` data to an Objective-C `NSString` using the complement of the technique we described for `databaseName`. `stringWithUTF8String` will convert our `char*` to an `NSString`, and we then assign the result to the `aName` variable. A similar process occurs for our other results rows.

From here, each `Food` object is added to our `Foods` array, and the process continues until our results are exhausted. We clean up by finalizing the compiled statement with `sqlite3_finalize()` and closing the database with `sqlite3_close()`. Note that we're only closing the database because we've completed the work we need to perform for our example. In a more complicated or featureful application, you may well leave the database open for further querying and choose to close it only under the `applicationWillTerminate` event.

Step 6: Final Polish and Wiring for iSeinfeld

There are several final wiring and configuration steps to have our main application launch the `FoodView` as required when viewing the details of our foods. Rather than include yet more code that isn't SQLite-specific, we'll refer you to the files in the included source code. We've liberally sprinkled comments through these files so you can see the changes we've made. You can also diff them against your vanilla files in Xcode to see these changes.

iSeinfeld in Action!

The suspense is over! We've written the code, worked with our SQLite database, and built a gorgeous user interface to show off at the next *Seinfeld* trivia night. You can now run your new application in the iOS simulator by choosing Build ➤ Build and Run. This will launch the device simulator, such as the iPhone Simulator shown in Figure 9-11.

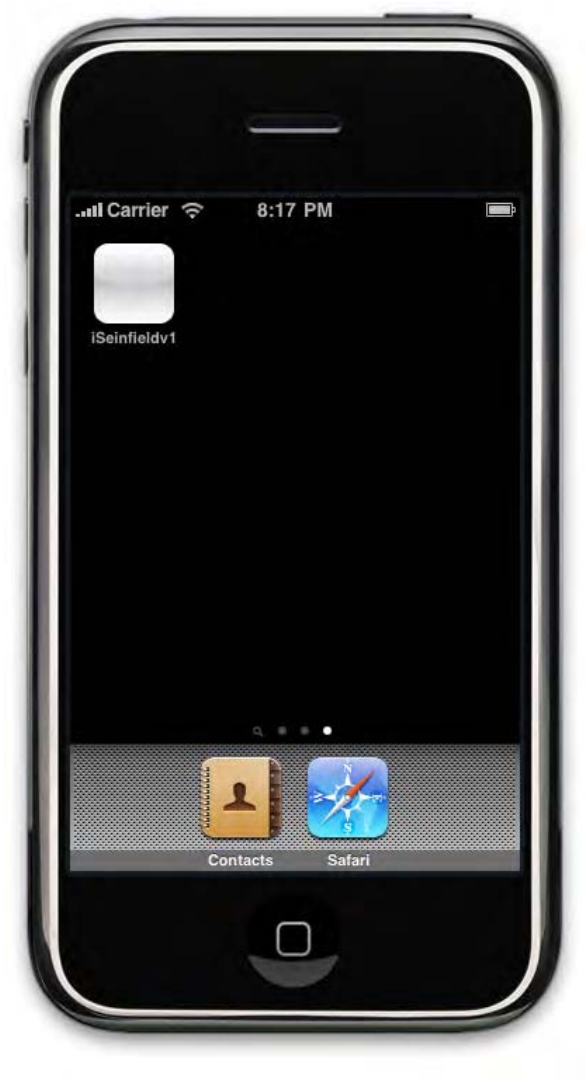


Figure 9-11. *The iSeinfeld SQLite-based application icon in the iOS device simulator*

Go ahead...click your new application to run it, and see for yourself what happens. Our main screen is shown next, in Figure 9-12.



Figure 9-12. *iSeinfeld main screen, showing foods from our* `foods.db` *SQLite database*

■ **Tip** If you want to watch the previous SQLite API calls step through their various tasks, set some breakpoints in the `readFoodsFromDatabase` function within `iSeinfeldAppDelegate.m`.

Choose any one of the foods listed, and the `FoodView` will kick in, controlled by our `FoodViewController`, to display the details of the food in question. In Figure 9-13, we see the details for Hot Dog—did you know it was the most popular food shown throughout the history of *Seinfeld*?



Figure 9-13. *iSeinfeld* `FoodView` *screen, showing the details for Hot Dog from the database*

Now, we'll be the first to admit that our graphic design skills don't set the world on fire. But the key to this application is you can see the data from your SQLite database brought to life through devices like the iPad and iPhone. Our journey with iSeinfeld is at an end. Go ahead and see how many enhancements you can make to your version of the application.

LEARNING ABOUT CORE DATA

This is a book about SQLite and using it for all kinds of problems. But there are times when you really want to abstract away the SQLite C API and deal with some more high-level abstraction that worries about all of the SQLite details for you. We've intentionally steered clear of Core Data—the data management abstraction layer available from iOS version 3 onward. We didn't do this because we wanted to tease you or make your life more difficult. Rather, we wanted to show you the heart of working with SQLite under iOS, rather than hiding it away.

For those of you interested in more abstract management of your SQLite database access and management under iOS, you can check out the Core Data documentation at `http://developer.apple.com/library/ios/`.

Working with Large SQLite Databases Under iOS

The eagle-eyed will have noticed that the query that derives our food frequency for the iSeinfeld app includes the following clause:

```
limit 50
```

Why limit this result set? We did this to simplify our application and avoid dealing with operating large SQLite databases and datasets in our very first example under iOS. If we'd attempted to do large database operations during the `applicationDidFinishLaunching` event, we ran the risk of incurring the wrath of the Watchdog Timer. The job of the Watchdog Timer is to ensure that all applications running on the iPad, iPhone, or iPod behave themselves. This includes preventing poorly performing applications from appearing to hang the device or consume too much processing time for a given action.

Apple describes the role of the Watchdog Timer in the Apple iOS Developer documentation as a system monitor that checks to see how long an application takes to complete its initial startup (among other things). If the application takes too long, the Watchdog Timer will terminate the offending application with exception code 0x8badf00d (get it, "bad food"). Related information will be written to the crash report for the incident.

When you launch your code in Xcode, the Watchdog Timer is disabled so as to compensate for the various overheads that our OS X system will incur, such as invoking debuggers, and so on. This means you may not notice the effect of the Watchdog Timer under Xcode simulation. It's always best to test on real devices to see whether you're running into long-running action issues.

This really boils down to a few common-sense management approaches for working with SQLite applications under iOS:

- Don't perform time-consuming queries, or load large databases, during an startup event such as `applicationDidFinishLaunching`.
- When running queries that return large results sets or that will take considerable time, consider breaking them up into digestible pieces of work
- Explore the more advanced iOS API features for managing long-running tasks.

At the end of the day, if the Watchdog Timer decides to intervene, you'll see the following (or similar) error:

```
The application iSeinfeld quit unexpectedly

Mac OS X and other applications are not affected
```

Under a real iPad or iPhone, instead of this error message, you will see the application attempt to load the main view and abort partway through. You will usually be returned to the home screen. Armed with this knowledge, you should be able to keep the Watchdog Timer a happy hound.

Summary

You've now experienced the trials and tribulations of creating your first (or latest) SQLite-based iOS application. We know your friends and family will be amazed at your newfound trivia knowledge. In all honesty, they'll probably be interested in just how easy it is to build your own iPad or iPhone application with SQLite.

If you do decide to make it big with your iSeinfeld application, be sure to let us know. We'd love to buy a copy of the application from iTunes to add to our collection of iSeinfeld variants.

CHAPTER 10

Android Development with SQLite

Android has stormed the mobile device market in the few short years it has been available, and if you've given in to temptation and bought a mobile phone, tablet, or other device running Android, then you've been a proud user of SQLite all this time! SQLite is one of the built-in data persistence technologies included in the Android platform and supported by the Dalvik virtual machine. Unlike iOS described in Chapter 9, the Java foundations of Dalvik mean that a purpose-built API wrapper is used around the underlying SQLite C API when application developers want to employ SQLite in their applications.

In this chapter, we'll mix things up with a walk-through of the Android SQLite library, its classes, and its methods. Instead of just a dry rehash of the documentation, we'll take the opportunity to build a working Android SQLite-based application (just as we did for iOS in Chapter 9). We'll use the concept of the *Seinfeld* food trivia application and walk through creating the code and UI for the application.

We'll also cover some of the developer considerations required for successfully maintaining Android applications on a device over time.

Prerequisites for SQLite Android Development

To do any development work with SQLite for Android, you'll need to equip yourself with the required tools. Once equipped with these tools, you'll be in a position to do all manner of Android development, not just work involving SQLite. Obviously, SQLite is our area of interest, so that's what we will focus on.

The four steps you need to complete are as follows:

1. Check and install minimum prerequisites, including the Java development kit (JDK).
2. Download and install the Android SDK starter package.
3. Download and install the Android Developer Tools (ADT) if supported by your integrated development environment (IDE).
4. Add any desired target Android platforms and other components to your environment.

We'll cover each of these quickly and move on to the SQLite work pronto!

Check Prerequisites and the JDK

To develop any Android application, you'll need to check that your development computer meets a few simple system requirements. First, you must be running Linux, OS X, or Windows XP or newer. The Android team lists particular versions and related dependencies at `http://developer.android.com/sdk/requirements.html`. In addition to that, about 500MB of free disk space is required for the Android components, in addition to space required for your IDE.

The most important prerequisite is the JDK. Android development requires either JDK 5 or JDK 6. Note that the Java runtime environment (JRE) is not sufficient. You can source the latest JDK from the Oracle web site if you require it, or you can choose an IDE that incorporates the JDK.

All your development of SQLite Android applications will be done in an IDE. There are many to choose from , but the most popular, and best for beginners, is Eclipse. Using Eclipse will enable you to also use Google's Android Developer Tools, which we cover shortly. The Android team recommends using Eclipse version 3.4 or newer and choosing a "classic" or "Java-focused" release.

Downloading and Installing the Android SDK Starter Package

The Android SDK Starter Package is the core set of tools required for any Android development. Importantly, it's not a complete SDK. Rather, it acts as a baseline that also guides you to download additional required components.

Google makes downloading the Android SDK Starter Package very easy. Browse to the site `http://developer.android.com/sdk/index.html`, and you'll see the latest SDK versions available directly for download. At the time of writing, release 7 is the latest release. Depending on your choice of Linux, OS X, or Windows, you'll need one of the following packages:

- Linux: android-sdk_r07-linux_x86.tgz
- Mac OS X: android-sdk_r07-mac_x86.zip
- Windows: android-sdk_r07-windows.zip

With the SDK downloaded, you can now proceed to installation. Extract the contents of your .zip or .tgz file to a convenient location on your computer. For example, we're using Eclipse under Windows, so we'll place the extracted SDK in the directory `c:\eclipse\dropins\android-sdk-windows`.

Once you've placed the extracted SDK in your desired location, you'll need to add that directory to your system's `PATH` environment variable. Refer to Chapter 2 if you need a refresher on how to add a directory to your `PATH`.

Downloading and Installing the Android Developer Tools

If you choose Eclipse as the IDE for your SQLite-based Android development, then you should download and install the Android Developer Tools in addition to the Android SDK Starter Package. ADT extends Eclipse to allow rapid setup of new Android projects through a dedicated Android *aspect* and provides additional tools for creating UIs, debugging, and exporting signed and unsigned packaged Android applications (called APKs).

To install ADT, run Eclipse, and choose Help ➤ Install New Software (that's under Eclipse 3.5—this may have a slightly different name in earlier Eclipse versions). Click Add to add a new repository, and enter the Android Developer Tools repository URL in the resulting Add Repository dialog box. Click OK, and return to the Install New Software screen. You should now see Developer Tools and the subcomponents Android DDMS and Android Developer Tools displayed, as shown in Figure 10-1.

Download from Wow! eBook <www.wowebook.com>

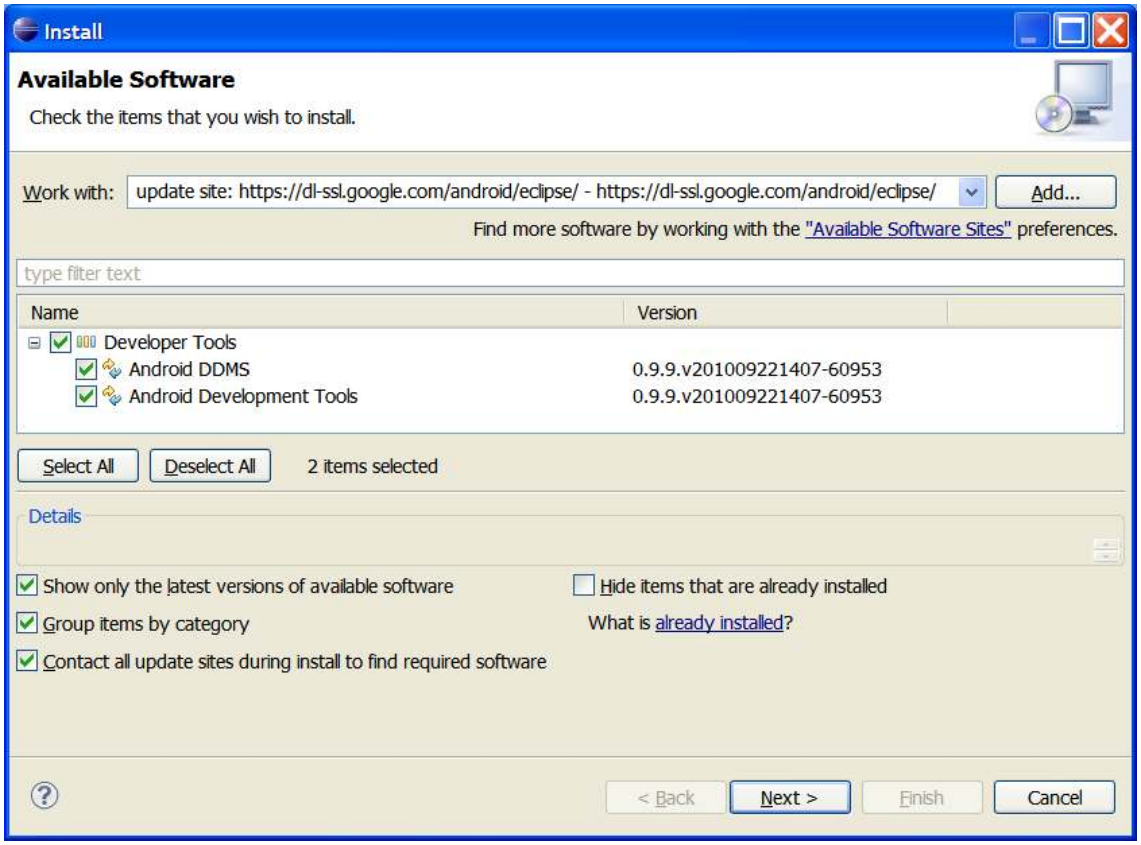


Figure 10-1. *Installing the Android Developer Tools*

Select all of the check boxes, and click the Next button. You'll be asked to agree to the licensing agreements and can then click Finish to complete the installation of ADT.

Adding Android Platforms and Components

The final task to get your Android SDK set up and ready for developing SQLite applications is to use the Android SDK and AVD Manager to download additional components to you environment. Google has built the full Android SDK to be modular, so things such as documentation, specific components for target versions and handsets, and other areas are split into separately managed and installed packages. You can choose the following components:

- *The SDK Tools*: This is the core part of the SDK and is already included in your environment when you install the SDK Starter Package.
- *Android Platforms*: An platform is available for each of the versions of the Android platform that have been released to production. You'll need to install at least one platform so you have a build target for your SQLite-based Android applications.
- *SDK Add-Ons*: The SDK Add-Ons are useful additional features bundled for use with the Android SDK. The most popular of these is the Google APIs Add-On, providing access to numerous Google technologies such as Maps, Street View, and so forth.
- *USB Driver for Windows*: The USB driver is provided for Windows developers so they can copy their applications to an actual device for running and debugging. Only Windows developers will need this—OS X and Linux need no special driver to allow this.
- *Samples*: This package includes numerous code snippets and examples of Android development.
- *Documentation*: This package contains full API documentation for all versions of the Android SDK.

Start the process of adding these components by choosing the menu option Window ➤ Android SDK and AVD Manager in Eclipse. The management window should appear, as shown in Figure 10-2.

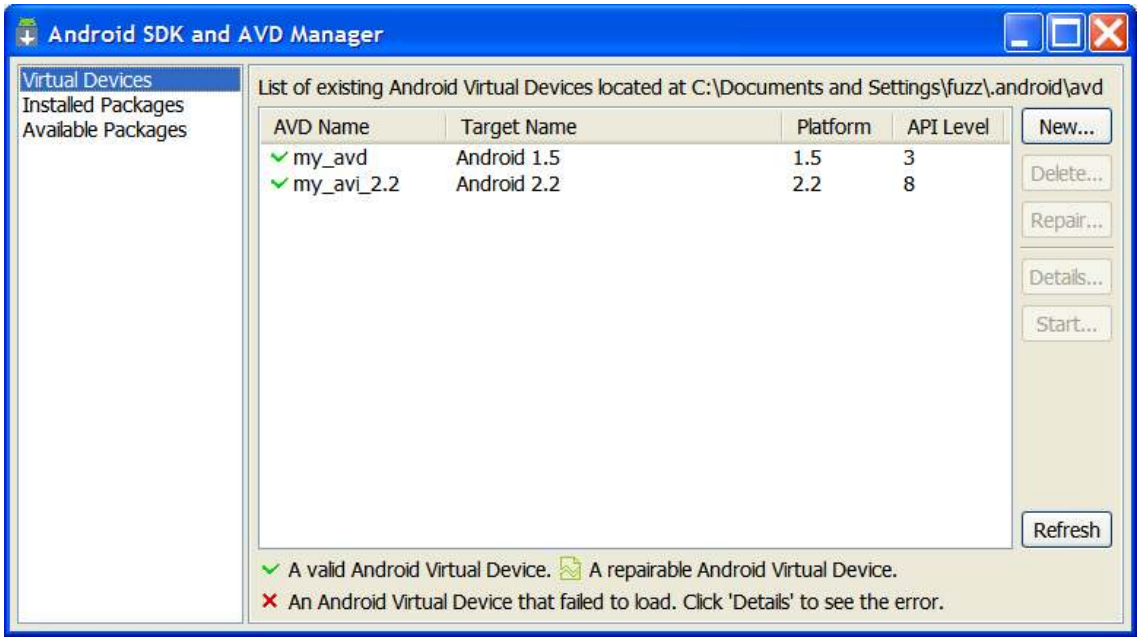


Figure 10-2. *The Android SDK and AVD Manager in Eclipse*

You may see one or more existing virtual devices listed in the manager. Let's create a new one specifically for our SQLite development, because this will allow us to ensure we have the components we want and need and none of the extraneous extras we don't need.

In the Android SDK and AVD Manager window, click the New button. You should see the Create New Android Virtual Device (AVD) dialog box appear, as shown in Figure 10-3.

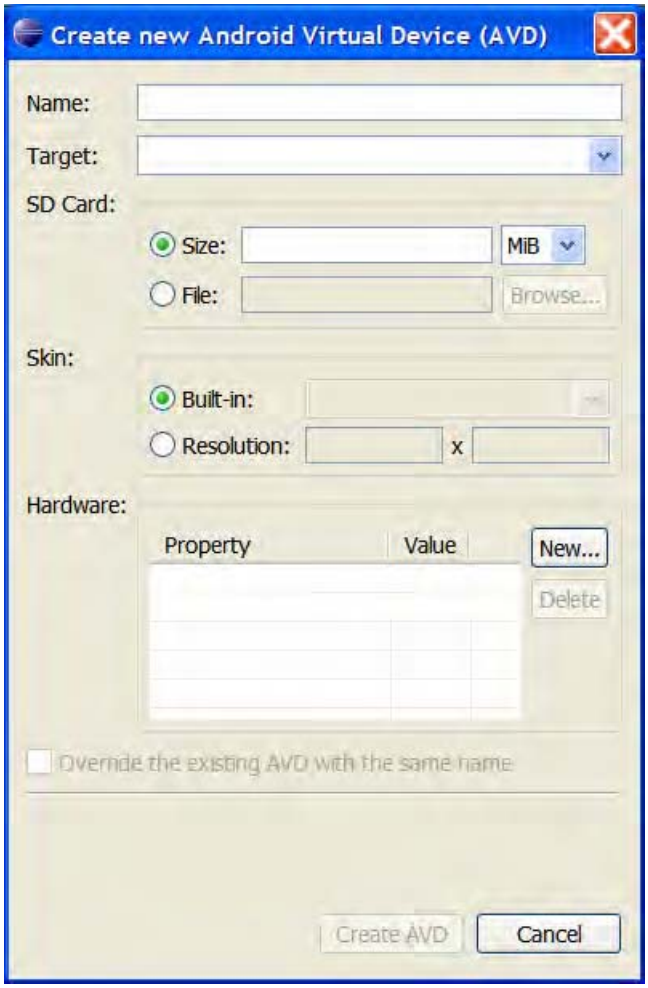


Figure 10-3. *Adding a new Android virtual device for the SQLite project*

You can call you virtual handset anything you like. We'll call ours **SQLite_AVD**. The Target field determines which Android SDK level you want to target for your application.

ANDROID VS. SQLITE VERSIONS

Where SQLite is concerned, there are slight variations in the underlying version of the SQLite libraries included. For Android releases up to and including version 2.1, the underlying SQLite version is 3.5.9. From Android release 2.2 onward (so-called Froyo), the SQLite library has been updated to version 3.6.22.

If you don't have any particular requirement in mind, you can support the oldest possible SDK release in order to target the widest possible audience. Otherwise, we recommend choosing at least the latest major release, which at this point is 2.2.

The SD Card and Skin settings aren't really a consideration with our SQLite focus, so at this point leave them at the defaults. The remaining aspect of your virtual device concerns the additional hardware this AVD will mimic. This includes many things that really won't impact our SQLite development, such as GPS and accelerometer devices. But several of the hardware options are worth considering.

The "Cache partition size" parameter will govern the fraction of on-board storage on the device available to applications and the phone's user (as opposed to Android and the system binaries). It's common to find phones and tablets with 4GB, 8GB, and even 16GB or more of onboard memory, and a good fraction of this is available to the Cache partition. If you plan to build applications with large SQLite databases, you should set this parameter to help you stay under the likely device ceilings your application would encounter in the wild.

The "Maximum VM application heap size" option sets the limit for the amount of heap allocation an Android application can take before it is killed by the system. This will impact your SQLite development when dealing with larger items allocated on the heap. The most likely culprits will be large result sets that are allocated as cursor objects (which we'll discuss later in this chapter). This defaults to 16MB. If you know you'll work with memory-limited devices, you can tweak this to enable your development to catch any low-memory scenarios before you reach real-world devices.

Device RAM Size designates the amount of physical memory present on the device. This is rarely going to be your limiting factor, but again for SQLite development on constraint devices, it's useful to dial down this value to make your development device more closely model your future target devices. The default is 96MB.

When you've finished choosing the hardware options you want for your virtual device, save it, and it should appear in the list of existing Android virtual devices in your environment.

ALTERNATIVES TO ECLIPSE FOR ANDROID DEVELOPMENT

For those experienced in Android development or looking for cross-platform mobile development tools, there are alternatives to Eclipse for Android development in general and SQLite-based Android development in particular. Feel free to use any or all of these if you're already comfortable with Android development and prefer them to Eclipse.

PhoneGap An open source development framework for building cross-platform mobile applications.
`www.phonegap.com`

Appcelerator Titanium Another open source cross-platform framework for mobile applications.
`www.appcelerator.com`

Each of these has excellent support for SQLite development under Android, as well as for other platforms.

The Android SQLite Classes and Interfaces

You now have a working environment in which to develop SQLite-based Android applications. The next step is to explore the classes and interfaces that Android uses to wrap the underlying SQLite C API. Unlike other APIs and language bindings that provide a one-to-one mapping between APIs, the Android SQLite API takes a very different approach.

Using the Basic Helper Class, SQLiteOpenHelper

The first and most important class provided by Android for working with SQLite databases is the `SQLiteOpenHelper` class, in the `android.database.sqlite` namespace. This is a helper class that is designed to be extended by you, to implement whatever tasks and actions you deem important when a database is first created, opened, or used. `SQLiteOpenHelper` has a single constructor defined as follows:

```
SQLiteOpenHelper(Context context, String name,
                 SQLiteDatabase.CursorFactory factory, int version)
```

The `context` is the application context allowing access to all the shared resources and assets for the given application. The `name` parameter contains the database file name within Android storage. The `factory` value introduces a new class, the `SQLiteDatabase.CursorFactory`. This is a factory class that generates `cursor` objects that act as the result set for all the queries you issue against SQLite under Android. The `version` parameter is your application-specific version number for the database (or more particularly, its schema). Don't confuse this with the actual SQLite version. `SQliteOpenHelper` will trigger its `onUpgrade()` method if your database isn't at the nominated version. All of `SQLiteOpenHelper`'s methods are as follows:

```
synchronized void close()
synchronized SQLiteDatabase getReadableDatabase()
synchronized SQLiteDatabase getWritableDatabase()
abstract void onCreate(SQLiteDatabase db)
void onOpen(SQLiteDatabase db)
abstract void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
```

The `close()` method holds no surprises. This method closes the SQLite database within the `SQLiteOpenHelper` object.

The next two methods, `getReadableDatabase()` and `getWriteableDatabase()`, perform similar actions, with one difference that you can probably guess from their names. The `getReadableDatabase()` method will open the database specified for the `SQLiteOpenHelper` object but will open it such that it is read-only. In effect, any data manipulation statement that attempts to change data will not be allowed. `getWriteableDatabase()` is used to open a SQLite database for normal read/write operations. If for some reason your database cannot be opened for write operations, `getWriteableDatabase()` will open the database as a read-only database (just as if `getReadableDatabase()` had been called) and will throw an exception of type `SQLiteException`. You can test a database's read or read/write state by calling the `isReadOnly()` method on the database object itself. This is a Boolean value that returns true for a read-only database. You can later call `getWriteableDatabase()` on a database that is read-only to reopen it as a read/write database. Calling either method on a database that doesn't yet exist will implicitly invoke the helper object's `onCreate()` method. Otherwise, the first call to either method from your application will invoke `onOpen()` and optionally `onUpgrade()`. Your database will then be cached for the application to use, until the `close()` method is called.

■ **Caution** Opening a large database can take some time, especially if you have complex logic built into your `onOpen()` or `onCreate()` implementations. We strongly recommend that for real-world applications you don't call this from the application's main thread at startup time.

The final three methods—`onCreate()`, `onOpen()`, and `onUpgrade()`—are designed for you to subclass to implement your desired behavior. `onCreate()` is triggered when the database is initially created and typically used to create tables and load data with insert statements to bring a fresh database into existence.

`onOpen()` is triggered when the database completes opening. You would typically check things such as the read/write status of the database at this point with `isReadOnly()` to ensure your database is in a known state before working with it.

The `onUpgrade()` method is called when the database needs to be upgraded from an application perspective (remember, we're not talking about SQLite versions here; we're talking about your own application version). You typically start numbering your database version at 1 and increment this to 2, then 3, and so on, as you release new versions of your application. You code your desired `alter table`, `create table`, `drop table`, and other statements within this method to handle graceful upgrades of the database schema for your application as it evolves.

Working with the SQLiteDatabase Class

Now that you are familiar with the helper class that kick-starts the use of SQLite databases within Android, it's time to look at the core `SQLiteDatabase` class. A `SQLiteDatabase` object is conceptually easy to understand, being very similar to the underlying database object in the SQLite C API. However, the implementation of a wide range of helper methods and other tools means there's a little more under the wrapper than you might expect.

More than 50 methods are available for the `SQLiteDatabase` class, each with its own nuances and use cases. Many of these are basic helper methods to easily complete simple tasks such as one-table select, insert, update, and delete statements. Rather than an exhaustive list, we'll cover the most important subsets of methods and allow you to explore some of the overloaded methods at your leisure. At any time you can refer to the full online Android documentation for the SQLiteDatabase class at `http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html`

Opening and Closing the SQLiteDatabase

`SQLiteDatabase` offers three methods to open a database and one to close a database. These methods are as follows:

```
void close()
static SQLiteDatabase openDatabase(String path,
                                   SQLiteDatabase.CursorFactory factory, int flags)
static SQLiteDatabase openOrCreateDatabase(File file,
                                   SQLiteDatabase.CursorFactory factory)
static SQLiteDatabase openOrCreateDatabase(String path,
                                          SQLiteDatabase.CursorFactory factory)
```

The `close()` method is as simple as it looks. The database is closed, and no further queries can be performed against the `SQLiteDatabase` object (that is, the database).

The `openDatabase()` method is the most versatile. It takes a `path` value to the file in your Android application's assets, together with an associated `factory` parameter that will act as a `CursorFactory` for your database. The final parameter is a "stackable" `flags` parameter that dictates options to opening the database. Four flag values pertain to opening the database.

- `OPEN_READWRITE`: Opens the database for reading and writing.
- `OPEN_READONLY`: Opens the database for reading (select) only.
- `CREATE_IF_NECESSARY`: Creates the database file first if it does not already exist.
- `NO_LOCALIZED_COLLATORS:` Opens the database without support for the `LOCALIZED` SQLite collations. These are custom collations that match the regional settings of the Android device.

The first `openOrCreateDatabase()` method accepts a `File` object as the target database to open. It does not have a `flags` parameter, defaulting to the behavior of the `CREATE_IF_NECESSARY` value.

The second `openOrCreate Database()` overloaded method is a shorthand version of `openDatabase()`. It acts in almost exactly the same fashion, accepting a `string` as the path to find the database file. It also has no flags parameter and likewise defaults to the `CREATE_IF_NECESSARY` behavior.

Executing General Queries with SQLiteDatabase

Just as there are numerous ways to compose SQL statements, Android provides a wealth of ways to run SQL against your SQLite database. In fact, there are no fewer than 16 methods that run general or specific styles of queries against your SQLite database. The can be separated into convenience functions that do single-table inserts, updates, and so forth, as well as general methods for executing DML and DDL. The general methods can be grouped together as shown in this list:

```
void execSQL(String sql)
void execSQL(String sql, Object[] bindArgs)
Cursor query(boolean distinct, String table, String[] columns, String selection,
             String[] selectionArgs, String groupBy, String having, String orderBy,
             String limit)
Cursor query(String table, String[] columns, String selection, String[] selectionArgs,
             String groupBy, String having, String orderBy)
Cursor query(String table, String[] columns, String selection, String[] selectionArgs,
             String groupBy, String having, String orderBy, String limit)
Cursor queryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, boolean distinct,
                        String table, String[] columns, String selection,
                        String[] selectionArgs, String groupBy, String having,
                        String orderBy, String limit)
Cursor rawQuery(String sql, String[] selectionArgs)
Cursor rawQueryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, String sql,
                           String[] selectionArgs, String editTable)
```

Don't be put off by the number of methods there. You can see that there are basically three core types—`execSQL()`, `query()`, and `rawQuery()`, with the latter two having variants that use the `CursorFactory` from your `SQLiteOpenHelper()` invocation if you choose to use it.

The two forms of `execSQL()` take a SQL statement as the `sql` parameter, and the second variant also accepts an array of bind parameters to bind to the query (`bindArgs`). The `execSQL()` methods are used for statements that don't return results, such as `create table`, `insert`, `update`, `alter table`, and so forth. The `void` method return type should help you remember this.

The `query()` and `queryWithFactory()` methods are essentially functions for performing lightweight single-table select statements against your database. Take a look at the various parameters. You'll see that they include `table`, `columns`, `orderBy`, and so on. In essence, each of these methods allows you to pass the clauses of a SQL statement to the relevant method, without having to include the SQL keywords themselves.

The final pair of methods, `rawQuery()` and `rawQueryWithFactory()`, allow you to use any string as a SQL select statement, returning the results as a `Cursor` object. You'll notice that each accepts an array of `Strings` named `selectionArgs`. Through this parameter, the `SQLiteDatabase` object will replace all question mark (`?`) bind parameters in your SQL statement with the `String` values in the array, on a positional basis (that is, the first question mark is replaced with the first element of the array, and so on).

THE STRANGENESS OF COMPILESTATEMENT

We've avoided discussing one method of `SQLiteDatabase`. That method is the `compileStatement()` method. You might think from its name that it would be a very useful, generic statement compilation and query preparation tool. In some respects it is, but in others, it is a little strange. First, here's the method signature:

```
public SQLiteStatement compileStatement (String sql)
```

Nothing too strange there. Take a `String` holding the SQL, and return an object of type `SQLiteStatement`. In fact, as the name suggests, this method compiles a statement and allows it to be reused (as per the reset semantics for the underlying SQLite C API). But then the strangeness kicks in.

`compileStatement()`, and for that matter the `SQLiteStatement` object to which it is related, can be used only on statements that return a maximum of one result row, with a maximum of one result column. In effect, the results of a `compileStatement()` call and the data member of a `SQLiteStatement` must be a simple scalar value. We can think of situations where this would be useful, such as counting qualifying rows in a table or result set. But we can think of a very much larger set of circumstances where you want your SQL to work with result *sets*!

So, now you know about `compileStatement()` and `SQLiteStatement`, but if you're like us, you'll find yourself rarely using it.

Using Convenience Methods with SQLiteDatabase

We mentioned 16 methods that can execute queries, but the preceding section covered only 8 methods. The other 8 methods are convenience methods that allow you to issue insert, update, delete, and replace statements against a single database table. These methods are as follows:

```
int delete(String table, String whereClause, String[] whereArgs)
long insert(String table, String nullColumnHack, ContentValues initialValues)
long insertOrThrow(String table, String nullColumnHack, ContentValues initialValues)
long insertWithOnConflict(String table, String nullColumnHack,
                          ContentValues initialValues, int conflictAlgorithm)
long replace(String table, String nullColumnHack, ContentValues initialValues)
long replaceOrThrow(String table, String nullColumnHack, ContentValues initialValues)
int update(String table, ContentValues values, String whereClause, String[] whereArgs)
int updateWithOnConflict(String table, ContentValues values, String whereClause,
                         String[] whereArgs, int conflictAlgorithm)
```

At first glance, the various convenience methods are very easily understood. Decide on the action you want to take, nominate the table with the appropriate parameter, build a `whereClause`, and provide `whereArgs`. Take a closer look at the `insert()` and related methods and, for that matter, the `replace()` methods as well. You'll see a rather strange `String` parameter, `nullColumnHack`. This works in conjunction with the `initialValues ContentValue` map. Where the `initialValues` map is empty, `nullColumnHack` will provide `NULL` values for columns to prevent the attempt to insert no data, which would naturally fail.

Although these methods at first seem attractive, they are principally designed to appeal to the object-oriented developer who prefers simple CRUD-style primitive data manipulation. The methods tend to support using the database as just a persistence layer and, as already mentioned, leads to failing to do justice to both the power and the elegance of a relational database like SQLite. Now that we've mentioned these methods, we'll move on in the hopes you'll use the power of the other methods for building and executing queries.

Managing Transactions with SQLiteDatabase

All of SQLite's transaction management fundamentals are respected by the Android wrapper. Several useful methods are available to start, end, and manage your transactions.

```
void    beginTransaction()
void    beginTransactionWithListener(SQLiteTransactionListener transactionListener)
void    endTransaction()
boolean    inTransaction()
void    setTransactionSuccessful()
```

Several of these methods come close to needing no explanation. `beginTransaction()` starts a SQLite transaction, and `endTransaction()` ends the current transaction context for the `SQLiteDatabase` object. Crucially, whether the transaction commits or rolls back is dependent on the transaction being marked as "clean." This is achieved by calling the `setTransactionSuccessful()` method. This added step is at first an annoyance, but in reality it exists to ensure you double-check all changes to the database before committing. The `setTransactionSuccessful()` method with throw an `IllegalStateException` exception if you are not in a transaction or have already set the transaction to successful. The `inTransaction()` method tests whether you are currently in an active transaction, returning `true` if that's the case.

This leaves the `beginTransactionWithListener()` method. This takes a `SQLiteTransactionListener` object as a parameter. An event will fire notifying this object every time a transaction management even happens in the transaction, whether that's a commit, rollback, or nested begin event.

Using Other SQLiteDatabase Methods

Numerous other helpful methods are available to the `SQLiteDatabase` object. Here are a select few to round out our discussion.

- `public long getMaximumSize()`: Returns the maximum size allowed for the database.
- `public int getVersion()`: Returns your application-specific database version.
- `public boolean isDbLockedByCurrentThread()`: Tests whether your current thread holds the database lock.
- `public boolean isDbLockedByOtherThreads()`: Tests whether another thread holds the database lock.
- `public static int releaseMemory()`: Releases all working memory no longer needed by the database, cursors, and so on. The number of bytes freed is returned.

There are a number of other simple Boolean methods to test various aspects of your database's state, as well as a number of methods to control synchronizing cached copies of your database back to storage and other tasks.

Applying SQLiteOpenHelper and SQLiteDatabase in Practice

We've now covered enough of the Android SQLite API to walk through an example implementation of our own derived `SQLiteOpenHelper` class complete with overridden methods to manage our `SQLiteDatabase`—your underlying SQLite database.

Listing 10-1 shows a sample (large, working) fragment of Android code that implements our own subclass of `SQLiteOpenHelper`, dealing with the common creation and opening tasks of using our *Seinfeld* database. This code is included as `myDatabaseHelper.java` in the sample code.

Listing 10-1. *`myDatabaseHelper.java`*

```
import java.io.*;
import android.database.sqlite.*;
import android.database.SQLException;
import android.content.Context;

public class myDatabaseHelper extends SQLiteOpenHelper{
  //DBPATH uses the default system path for a given application
  // /data/data/<app namespace> , which in our example will be com.example.seinfeld
  private static String DBPATH = "/data/data/com.example.seinfeld/databases/";
  private static String DBNAME = "foods.db";
  private SQLiteDatabase myDatabase;
  private final Context myContext;
```

Download from Wow! eBook <www.wowebook.com>

```
//constructor
public myDatabaseHelper(Context context) {
  super(context, DBNAME, null, 1);
  this.myContext = context;
}

//create an empty db, and replace with our chosen db
public void createDatabase() throws IOException{
  if (!checkDatabase()) {
    this.getWritableDatabase();
    try {
      copyDatabase();
    }
    catch (IOException e) {
      throw new Error("Error copying database from system assets");
    }
  }
}

//Check if our database already exists
private boolean checkDatabase(){
  SQLiteDatabase checkableDatabase = null;
  try {
    checkableDatabase =
      SQLiteDatabase.openDatabase(DBPATH+DBNAME, null,
                                  SQLiteDatabase.OPEN_READONLY);
  }
  catch (SQLiteException e) {
    //our database doesn't exist, so we'll return false below.
  }
  if (checkableDatabase != null) {
    checkableDatabase.close();
  }
  return checkableDatabase != null ? true : false;
}

//Copy our database from the Application's assets
//over the empty DB for use
private void copyDatabase() throws IOException{

  InputStream myInput = myContext.getAssets().open(DBNAME);
  OutputStream myOutput = new FileOutputStream(DBPATH+DBNAME);

  byte[] buffer = new byte[1024];
  int length;
  while ((length = myInput.read(buffer))>0){
    myOutput.write(buffer, 0, length);
  }
```

```
    myOutput.flush();
    myOutput.close();
    myInput.close();
  }

  public void openDatabase() throws SQLException{
    myDatabase = SQLiteDatabase.openDatabase(DBPATH+DBNAME, null,
                                             SQLiteDatabase.OPEN_READWRITE);
  }

  @Override
  public synchronized void close() {
    if(myDatabase != null)
      myDatabase.close();
    super.close();
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    //Handle creation tasks, etc.
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    //Handle upgrade tasks, etc.
  }
} //end of myDatabaseHelper.java
```

We can now use our `myDatabaseHelper` class to open or create our *Seinfeld* foods database.

```
…
myDatabaseHelper mySeinfeldDBHelper = new myDatabaseHelper();
mySeinfeldDBHelper = new myDatabaseHelper(this);

try {
  mySeinfeldDBHelper.createDatabase();
}
catch (IOException e) {
  throw new Error("Failed to create Seinfeld database");
}

try {
  mySeinfeldDBHelper.openDatabase();
}
catch(SQLException e) {
  throw e;
}
…
```

We'll use our `myDatabaseHelper` class toward the end of the chapter to help build our working Android SQLite-based application.

Querying SQLite with SQLiteQueryBuilder

In the fine tradition of SQLite APIs, the Android SQLite class libraries support both database-style objects and statement-style objects issuing queries against the database. We've already seen the `SQLiteDatabase` object in action. Now we'll look at the complementary class, `SQLiteQueryBuilder`.

Just as with `SQLiteDatabase`, `SQLiteQueryBuilder` offers a range of methods to help compose a valid SQL statement, without the need to provide the various SQL reserved words for your clauses. Its key difference is the way the methods provide features to directly execute the generated statement against a provided `SQLiteDatabase` object, as well as being able to emit a string to pass to a `SQLiteDatabase` object for execution under that object's auspices. First, let's take a look at the methods supporting direct execution of your generated statement:

```
Cursor query(SQLiteDatabase db, String[] projectionIn, String selection,
             String[] selectionArgs, String groupBy, String having, String sortOrder)
Cursor query(SQLiteDatabase db, String[] projectionIn, String selection,
             String[] selectionArgs, String groupBy, String having, String sortOrder,
             String limit)
```

Those methods look eerily familiar. They are almost exactly identical to the `query()` methods of the `SQLiteDatabase` object, with a few exceptions. Rather than invoking them on the database object itself, you pass the database object as the first parameter (`db`). Many people find this just a case of personal preference, but there are use cases where the `SQLiteQueryBuilder` approach is helpful. You'll note the only difference between the two overloaded methods is that the second allows for a `limit` clause.

But wait! What's that? The careful reader will have spotted one more difference. Where has our `table` parameter gone? This is one of the key differences when using `SQLiteQueryBuilder`. A separate method is used to specify the table or tables against which the query will operate. This is the `setTables()` method.

```
public void setTables (String inTables)
```

■ **Note** This is probably the most common "Gotcha" developers encounter when switching between `SQLiteQueryBuilder` and `SQLiteDatabase`.

The `setTables()` method simply takes a comma-separated string of table names as a parameter. There are a few additional methods of `SQLiteQueryBuilder` that operate this way. These include `setDistinct()`, for indicating the SQL should use the `distinct` keyword, and `setProjectionMap (Map<String, String> columnMap)`, which controls the aliasing of columns and column disambiguation.

The bulk of the remaining methods for `SQLiteQueryBuilder` are the "build" methods. These methods are designed to build a query string and return it for use elsewhere, such as through a `SQLiteDatabase` object or recursively in the `SQLiteQueryBuilder` itself.

```
String buildQuery(String[] projectionIn, String selection, String[] selectionArgs,
                  String groupBy, String having, String sortOrder, String limit)
static String buildQueryString(boolean distinct, String tables, String[] columns,
                               String where, String groupBy, String having,
                               String orderBy, String limit)
String  buildUnionQuery(String[] subQueries, String sortOrder, String limit)
String  buildUnionSubQuery(String typeDiscriminatorColumn, String[] unionColumns,
                           Set<String> columnsPresentInTable, int computedColumnsOffset,
                           String typeDiscriminatorValue, String selection,
                           String[] selectionArgs, String groupBy, String having)
```

Once again, there's a great deal of similarity there with the methods seen from `SQLiteDatabase`. Note however, that there are methods that focus heavily on union queries. Personally, we find it easier to construct these ourselves and use the `rawQuery()` methods on `SQLiteDatabase`, but if you're programmatically building up your union statements, the approach of `buildUnionQuery()` and the related methods provided added resilience against SQL injection attacks.

That's enough theory on the Android SQLite class libraries. Now let's move on to create a working Android SQLite-based application.

Building the *Seinfeld* Android SQLite Application

With a working development environment now in place from the start of this chapter and a working knowledge of the Android SQLite class libraries, you are ready to build your SQLite-based Android application. Our requirements are very similar to the iOS application we built in Chapter 9:

- Build an Android application that shows the foods mentioned in *Seinfeld* by order of popularity
- For any popular food, show how often the food is mentioned over all series of *Seinfeld*
- Indicate the show in which a food first appeared

Once again, we have a very simple set of requirements. We'll use the `myDatabaseHelper.java` class already developed earlier in the chapter, together with a little Android UI magic. That means some short coverage of non-SQLite areas will be needed, but we'll keep that to a minimum.

Here are the steps we will follow:

1. Create a new Android project in Eclipse.
2. Add the SQLite framework to project.
3. Add our *Seinfeld* `foods.db` to our application assets.
4. Define the classes that map our food data from SQLite to our code.
5. Define the UI elements that present our summary and detail SQLite data on-screen.
6. Define our UI behavior.
7. Launch!

Let's begin.

Creating a New Android Project

Launch Eclipse to get started. Choose File ➤ New ➤ Android Project, and you should see the New Android Project dialog box appear, as in Figure 10-4.

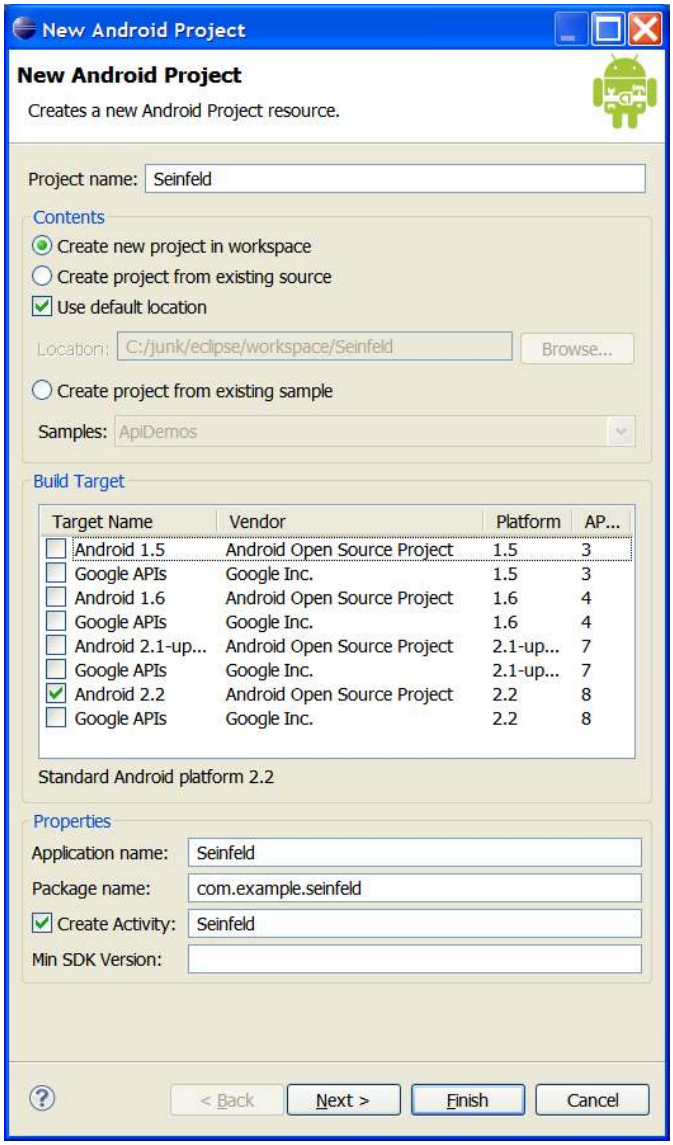


Figure 10-4. *Creating a new Android SQLite project*

Fill in the project name, application name, and package name, as shown in Figure 10-3, or use meaningful names that you're likely to remember. The application name will be the title that appears with your application on Android. Your package name should follow normal Java package naming—remember this needs to be globally unique.

Adding the Seinfeld SQLite Database to Your Project

To add the `foods.db` SQLite database file to your project, open the *Seinfeld* project folder, and then open the assets folder. This will be empty by default. You can either drag and drop the `foods.db` file from your operating system on to the assets folder or choose to link the database file in place. We don't recommend the link option, because this makes packaging your application with the database more difficult.

Later in the chapter we'll explore how the `adb` utility enables you to still use the command line against your `foods.db` database, even after it has been loaded onto your (virtual or real) handset.

Querying the Foods Table

We now need to add additional functionality to our *Seinfeld* application to query our database to return the foods we know and love. We also need to "wire up" the results to a `ListView` in the Android UI world so that we can see our results through the Android application and play around with them.

Our first task is to add a method to our `myDatabaseHelper` class to fetch all the foods from the `food` table. This fragment is from the `myDatabaseHelper.java` file available with the sample code:

```
…
public Cursor fetchAllFoods() {

  return myDatabase.rawQuery("select name from foods order by name", null);
}
…
```

As you can see, we're not attempting a huge amount of sophistication here. You can always revisit our example later to increase the complexity of your *Seinfeld* application.

Next, we need to feed the `Cursor` for all our fetched foods into the `ListView` that will display them. Within our `Seinfeld.java` code, we declare a few handy strings representing our known field names and then a method to iterate over a result set, binding returned fields and values to our `ListView`:

```
…
  public static final String FOODS_NAME_FIELD = "name";
...

private void fillData() {
  Cursor foodCursor = mDbHelper.fetchAllNotes();
  startManagingCursor(foodCursor);

  // Create an array for our food names
  String[] from = new String[]{myDatabaseHelper.FOODS_NAME_FIELD};
```

```
  // Create an array of fields for binding to
  int[] to = new int[]{R.id.text1};

  SimpleCursorAdapter foods =
    new SimpleCursorAdapter(this, R.layout.food_row, foodCursor, from, to);
  setListAdapter(foods);
}
```

Don't worry if the Android-specific UI tweaking doesn't make sense; we'll discuss the mysterious `R` object next and how the `text1` field is defined.

Defining the User Interface

When your fresh *Seinfeld* project was created, a blank canvas was put in place for the user interface. If you drill down into your project, under the `res` folder you'll see a `main.xml` file. This would be where you would start designing a fresh layout, either using a graphical interface builder or carefully crafting Android UI XML by hand.

Thankfully, we're not going to put you through that torture. Instead, we'll walk you through the example code we've provided. Listing 10-2 shows you the contents of our `food_list.xml` file. This controls the overall look and feel for our application as a whole.

Listing 10-2. *`food_list.xml`*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

 <ListView        android:id="@id/android:list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
        <TextView android:id="@id/android:empty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/no_foods"/>
</LinearLayout>
```

Very briefly, our design has a `LinearLayout` specification that is the most common for textual applications. Within that, we define a set of expected behaviors for our UI depending on what our UI is passed to display. Should a populated `ListView` be provided, we attempt to display this using list semantics. Should a (nominally empty) `TextView` be provided, we'll display the string `no_foods`. Otherwise, we'll display the textual content. Only one of these two views will be used at any one time. The values `list` and `empty` are helpers provided by the Android platform, to ease the complexity of defining the behavior of UI components.

This makes sense when viewed in conjunction with our other layout file, `food_row.xml`, shown in Listing 10-3.

Listing 10-3. *food_row.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/text1"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Here we're instructing Android that we'd like a new ID called `text1`. The use of the `+` symbol is shorthand to say "If I've forgotten to define `text1` by now, go ahead and do here, so I can start using it immediately." We find it handy to use this approach to keep the definition of UI elements as close as possible to where they'll be used.

Go ahead and include these UI XML files in your project. Through the magic of the Android Developer Tools, Eclipse will note your newly defined field and IDs and add the necessary values to the mysterious `R.java` file. Actually, there really isn't too much mystery here. The `R.java` file is managed by ADT and automatically connects any UI resource to your Java code. Feel free to inspect `R.java`, but you'll rarely if ever need to manually change it.

Linking the Data and User Interface

Our last major task is to control when and how our interface receives its data. For this exercise, we'll take the subclass of `onCreate()` that we implemented earlier in the chapter, expand it to find and open our SQLite foods database, and populate the `ListView` we've created with the results of the `fillData()` method we've built. This code fragment is included in the file `Seinfeld.java`.

```
…
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.food_list);
        mySeinfeldDbHelper = new MyDatabaseAdapter(this);
        mySeinfeldDbHelper.createDatabase();
        mySeinfeldDbHelper.openDatabase();
        fillData();
    }
…
```

Here, we invoke `setContentView` to accept our food `ListView`, `R.layout.food_list`. Next, we create an object using our `myDatabaseAdapter` class and create (copy to cache) and open the database. We then call `fillData()`, which from our earlier definition does the hard work of taking our `Cursor` full of food results and feeding the data into `TextView` elements for our `ListView`.

That's really about it. Be sure to take a look at the full listings for `Seinfeld.java` and the related code in the sample files included. We didn't want to waste a few dozen pages on printing code that you can more easily review yourself on-screen, but looking at the code in context will help further your understanding.

Viewing the Finished Seinfeld Application

We've walked through the new and changed code that makes up our Seinfeld application, so now it is time to see it in action. With Eclipse, choose Run ➤ Run As ➤ Android Application. This will launch your application in your Android virtual device created earlier in the chapter.

■ **Note** It can take a few minutes for the virtual device to initialize, load, and be ready to use. Even then, it can take a few seconds longer to see your application. You should also check that you've activated the screen lock pattern to ensure your *Seinfeld* application is patiently waiting behind the screen lock for you.

Once your *Seinfeld* application has started, you should see familiar food names from our trusty `foods.db` SQLite database, as shown in Figure 10-5.



Figure 10-5. *Your running* ***Seinfeld*** *Android SQLite application*

That's it, you've created your first Android SQLite application. From here, you can go crazy adding features and devising new applications.

Care and Feeding for SQLite Android Applications

As with all mobile platforms, the somewhat constrained nature of hardware and resources under Android (compared with normal desktop or notebook computers) means some additional care and attention is warranted to ensure the best possible development and user experience.

Database Backup for Android

You've seen in our development of the `myDatabaseHelper.java` subclass that it is possible to work directly with the SQLite database file. Using normal Android API calls, it is possible to manipulate the SQLite file yourself, and therefore you could use this approach to provide for backups and recovery of your database. But there is a better way.

Google provides a backup service with Android, which allows you to copy any kind of persistent data to remote storage. This applies not just to SQLite database files, but that's our focus for now. This is a very useful way to provide for data resilience, backup, and recovery, way beyond what you could achieve manually. For instance, if a user invokes a factory reset on an Android-powered device, a do-it-yourself backup would likely be lost. But the data backup system can automatically restore your data when your application is reinstalled.

This feature is quite in-depth and is based around a number of classes. `BackupAgent`, `BackupAgentHelper`, and `BackupManager` are the main APIs involved. As with some of the classes we've seen, each of these classes includes a number of methods and subclasses designed to be overridden by you so that you can provide application-specific behavior for backup and restore activities. There's enough material required to cover these tasks to fill a book in its own right, so we'll point you to the online documentation for further reading at `http://developer.android.com/guide/topics/data/backup.html`.

Download from Wow! eBook <www.wowebook.com>

Working with Large SQLite Databases Under Android

In our introduction to creating an Android virtual device, we mentioned several configuration options that control or mimic the memory, cache, and storage resources available to you when creating any Android application. These included maximum VM application heap size, cache partition size, and device RAM Size.

A subtle but frustrating issue can arise when you begin working with large sets of data derived from your SQLite databases. You run some queries, fetch very large results into your `Cursor` object, and start working with the data. At some point, your UI crashes and leaves you with bizarre UI errors in the trace output. What's going on?

Despite the sometime prodigious amounts of memory and storage now available on Android devices and configurable in your AVD, there are other limits that can crimp your style. The most common of these involves using the `CursorWindow` object as a secondary cache of `Cursor` result rows. A `CursorWindow` can accommodate only 1MB of data. The variable smooth operation or crashing behavior you can see can be caused by attempting to exceed the `CursorWindow` size limit with too many rows and or rows with excessively long data. Naturally, sometimes your data may be short and sweet, thus showing no problem, but at other times…kapow!

Our advice is to remember that you are dealing with devices usually sporting only a 3-inch or 4-inch screen and that caching thousands of rows is probably not a useful design. What user of yours will ever scroll or read through that amount of data in one action, especially on a screen that small?

Summary

You now have a firm understanding of the tools, classes, and practice of creating SQLite-based Android applications. The Android SQLite API is somewhat different from the APIs we've covered elsewhere, and it is evolving at a rapid rate, just as the rest of the Android platform is also evolving.

Because Android is proving to be a runaway mobile platform success, we're sure any interest you have in creating SQLite-based applications for Android will be well rewarded. Good luck!

CHAPTER 11

SQLite Internals and New Features

Our last chapter is a collection of topics devoted to the internals of SQLite. The material here is collected from topics raised by Richard Hipp, SQLite's founder, over the past year or so. Importantly, we'll highlight one of the biggest changes to happen to SQLite in recent versions: the introduction of the Write-Ahead Log model. We'll also briefly explore how the pager and B-trees work and go into a little more detail on the underpinnings of data types, affinity, and the SQLite approach to typing.

This isn't an exhaustive description of the inner workings of SQLite, because that's covered very well on the SQLite web site, and the healthy rate of change for SQLite releases means that a book isn't always the best place to track things such as known commands in the VDBE, optimization models, and so on. We hope you enjoy these topics, and we encourage you to read more to further your understanding of the beauty and elegance of SQLite's design.

The B-Tree and Pager Modules

The B-tree provides SQLite's VDBE with *O(logN)* lookup, insert, and delete as well as *O(1)* bidirectional traversal of records. It is self-balancing and automatically manages both defragmentation and space reclamation. The B-tree itself has no notion of reading or writing to disk. It concerns itself only with the relationship between pages. The B-tree notifies the pager when it needs a page and also notifies it when it is about to modify a page. When modifying a page, the pager ensures that the original page is first copied out to the journal file if the traditional rollback journal is in use. Similarly, the B-tree notifies the pager when it is done writing, and the pager determines what needs to be done based on the transaction state.

Database File Format

All pages in a database are numbered sequentially, beginning with 1. A database is composed of multiple B-trees—one B-tree for each table and index (B+trees are used for tables; B-trees are used for indexes). Each table or index in a database has a root page that defines the location of its first page. The root pages for all indexes and tables are stored in the `sqlite_master` table.

The first page in the database—Page 1—is special. The first 100 bytes of Page 1 contain a special header (the file header) that describes the database file. It includes such information as the library version, schema version, page size, encoding, whether autovacuum is enabled—all of the permanent database settings you configure when creating a database along with any other parameters that have been set by various pragmas. The exact contents of the header are documented in `btree.c` and at `http://www.sqlite.org/fileformat2.html`. Page 1 is also the root page of the `sqlite_master` table.

Page Reuse and Vacuum

SQLite recycles pages using a free list. That is, when all the records are deleted out of a page, the page is placed on a list reserved for reuse. When new information is later added, nearby pages are first selected before any new pages are created (expanding the database file). Running a `VACUUM` command purges the free list and thereby shrinks the database. In actuality, it rebuilds the database in a new file so that all in-use pages are copied over, while free-list pages are not. The end result is a new, compacted database. When autovacuum is enabled in a database, SQLite doesn't use a free list and automatically shrinks the database upon each commit.

B-Tree Records

Pages in a B-tree consist of B-tree records, also called *payloads*. They are not database records in the sense you might think—formatted with the columns in a table. They are more primitive than that. A B-tree record, or payload, has only two fields: a key field and a data field. The key field is the `rowid` value or primary key value that is present for every table in the database. The data field, from the B-tree perspective, is an amorphous blob that can contain anything. Ultimately, the database records are stored inside the data fields. The B-tree's job is order and navigation, and it primarily needs only the key field to do its work (however, there is one exception in B+trees, which is addressed next). Furthermore, payloads are variable size, as are their internal key and data fields. On average, each page usually holds multiple payloads; however, it is possible for a payload to span multiple pages if it is too large to fit on one page (for example, records containing `blob`s).

B+Trees

B-tree records are stored in key order. All keys must be unique within a single B-tree (this is automatically guaranteed because the keys correspond to the `rowid` primary key, and SQLite takes care of that field for you). Tables use B+trees, which do not contain table data (database records) in the internal pages. Figure 11-1 shows an example B+tree representation of a table.

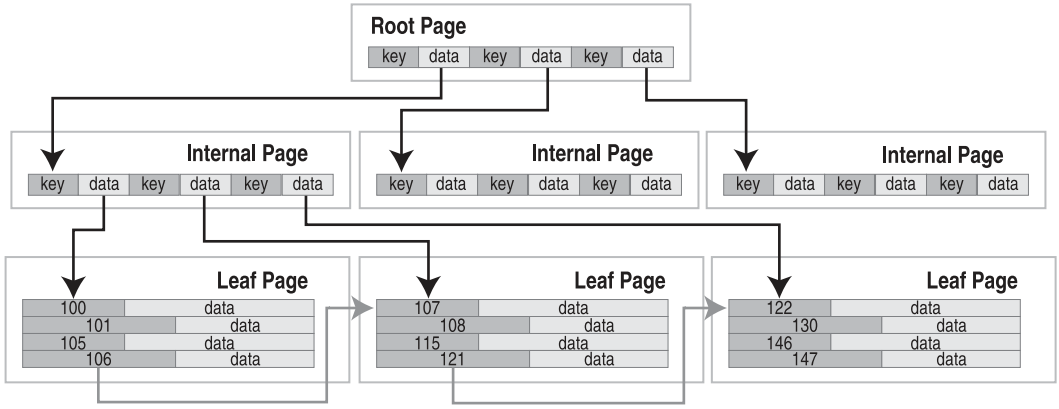


Figure 11-1. *B+tree organization (tables)*

The root page and internal pages in B+trees are all about navigation. The data fields in these pages are all pointers to the pages below them—they contain keys only. All database records are stored on the leaf pages. On the leaf level, records and pages are arranged in key order so it is possible for B-tree cursors to traverse records (horizontally), both forward and backward, using only the leaf pages. This is what makes traversal possible in O(1) time.

Records and Fields

The database records in the data fields of the leaf pages are managed by the VDBE. The database record is stored in binary form using a specialized record format that describes all the fields in the record. The record format consists of a continuous stream of bytes organized into a logical header and a data segment. The header segment includes the header size (represented as a variable-sized 64-bit integer value) and an array of types (also variable 64-bit integers), which describe each field stored in the data segment, as shown in Figure 11-2. Variable 64-bit integers are implemented using a Huffman code.

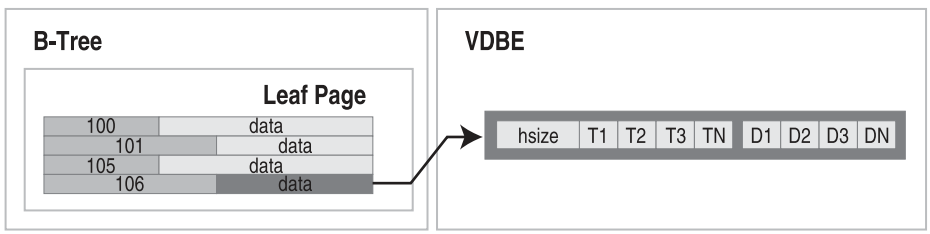


Figure 11-2. *Record structure*

The number of type entries corresponds to the number of fields in the record. Furthermore, each index in the type array corresponds to the same index in the field array. A type entry specifies the data type and size of its corresponding field value. Table 11-1 lists the possible types and their meanings.

Table 11-1. *Field Type Values*

Type Value	Meaning	Length of Data
0	`NULL`	0
N in 1..4	Signed integer	N
5 Signed	integer	6
6 Signed	integer	8
7 IEEE	float	8
8	Integer constant 0	0
9	Integer constant 1	0
10-11	Reserved for future use	N/A
N>12 and even	`blob`	(N-12)/2
N>13 and odd	`text`	(N-13)/2

For example, take the first record in the `episodes` table:

```
sqlite> SELECT * FROM episodes ORDER BY id LIMIT 1;
id   season  name
---  ------  --------------------
0    1       Good News Bad News
```

Figure 11-3 shows the internal record format for this record.



Figure 11-3. *First record in the* `episodes` *table*

The header is 4 bytes long. The header size reflects this and itself is encoded as a single byte. The first type, corresponding to the `id` field, is a 1-byte signed integer. The second type, corresponding to the `season` field, is as well. The `name` type entry is an odd number, meaning it is a text value. Its size is therefore given by (49-13)/2=17 bytes. With this information, the VDBE can parse the data segment of the record and extract the individual fields.

Hierarchical Data Organization

Basically, each module in the stack deals with a specific unit of data. From the bottom up, data becomes more refined and more specific. From the top down, it becomes more aggregated and amorphous. Specifically, the C API deals in field values, the VDBE deals in records, the B-tree deals in keys and data, the pager deals in pages, and the OS interface deals in binary data and raw storage, as illustrated in Figure 11-4.

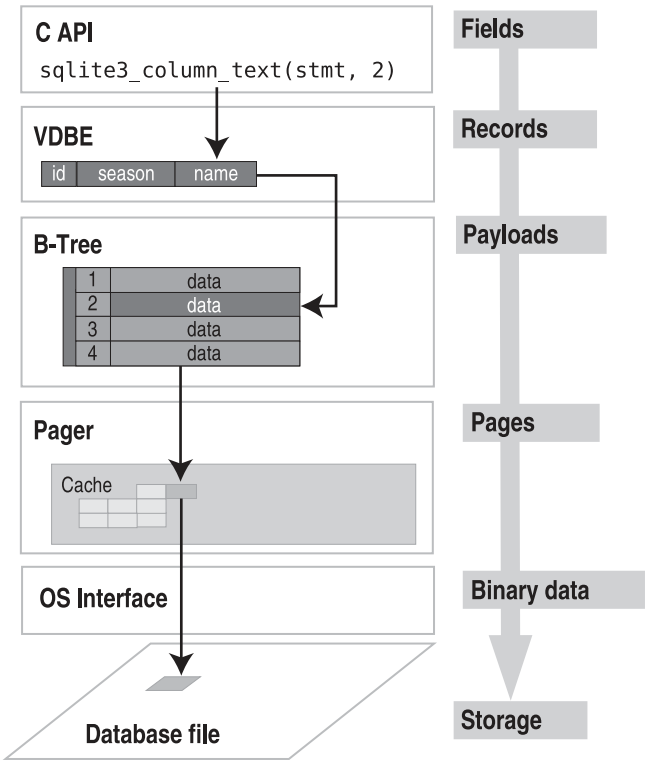


Figure 11-4. *Modules and their associated data*

Each module takes part in managing its own specific portion of the data in the database and relies on the layer below it to supply it with a cruder form from which to extract its respective pieces.

Overflow Pages

As mentioned earlier, payloads and their contents can have variable sizes. However, pages are fixed in size. Therefore, there is always the possibility that a given payload could be too large to fit in a single page. When this happens, the excess payload is spilled out onto a linked list of overflow pages. From this point on, the payload takes on the form of a linked list of sorts, as shown in Figure 11-5.

The fourth payload in the figure is too large to fit on the page. As a result, the B-tree module creates an overflow page to accommodate. It turns out that one page won't suffice, so it links a second. This is essentially the way binary large objects are handled. One thing to keep in mind when you are using really large `blob`s is that they are ultimately being stored as a linked list of pages. If the `blob` is large enough, this can become inefficient, in which case you might consider dedicating an external file for the `blob` and keeping this file name in the record instead.

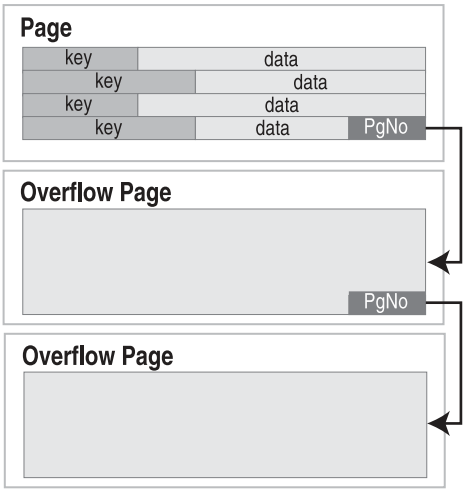


Figure 11-5. *Overflow pages*

The B-Tree API

The B-tree module has its own API, which is separate from the C API. We'll explore it here for completeness, but it's not really meant as a general-purpose B-tree API. You'll see it is heavily tailored to the needs of SQLite, so it's not really suitable for you to just pick up and drop into some other project. That said, understanding it in depth will allow you to appreciate even more of SQLite's internals. An added benefit of SQLite's B-tree module is that it includes native support for transactions. Everything you know about the transactions, locks, and journal files are handled by the pager, which serves the B-tree module. The API can be grouped into functions according to general purpose.

Access and Transaction Functions

Database and transaction routines include the following:

- `sqlite3BtreeOpen`: Opens a new database file. Returns a B-tree object.
- `sqlite3BtreeClose`: Closes a database.
- `sqlite3BtreeBeginTrans`: Starts a new transaction.
- `sqlite3BtreeCommit`: Commits the current transaction.

- `sqlite3BtreeRollback`: Rolls back the current transaction.
- `sqlite3BtreeBeginStmt`: Starts a statement transaction.
- `sqlite3BtreeCommitStmt`: Commits a statement transaction.
- `sqlite3BtreeRollbackStmtsqlite3BtreeRollbackStmt`: Rolls back a statement transaction.

Table Functions

Table management routines include the following:

- `sqlite3BtreeCreateTable`: Creates a new, empty B-tree in a database file. Flags in the argument determine whether to use a table format (B+tree) or index format (B-tree).
- `sqlite3BtreeDropTable`: Destroys a B-tree in a database file.
- `sqlite3BtreeClearTable`: Removes all data from a B-tree but keeps the B-tree intact.

Cursor Functions

Cursor functions include the following:

- `sqlite3BtreeCursor`: Creates a new cursor pointing to a particular B-tree. Cursors can be either a read cursor or a write cursor. Read and write cursors may not exist in the same B-tree at the same time.
- `sqlite3BtreeCloseCursor`: Closes the B-tree cursor.
- `sqlite3BtreeFirst`: Moves the cursor to the first element in a B-tree.
- `sqlite3BtreeLast`: Moves the cursor to the last element in a B-tree.
- `sqlite3BtreeNext`: Moves the cursor to the next element after the one it is currently pointing to.
- `sqlite3BtreePrevious`: Moves the cursor to the previous element before the one it is currently pointing to.
- `sqlite3BtreeMoveto`: Moves the cursor to an element that matches the key value passed in as a parameter. If there is no match, leaves the cursor pointing to an element that would be on either side of the matching element, had it existed.

Record Functions

Key and record functions include the following:

- `sqlite3BtreeDelete`: Deletes the record to which the cursor is pointing
- `sqlite3BtreeInsert`: Inserts a new element in the appropriate place of the B-tree
- `sqlite3BtreeKeySize`: Returns the number of bytes in the key of the record to which the cursor is pointing
- `sqlite3BtreeKey`: Returns the key of the record to which the cursor is currently pointing
- `sqlite3BtreeDataSize`: Returns the number of bytes in the data record to which the cursor is currently pointing
- `sqlite3BtreeData`: Returns the data in the record to which the cursor is currently pointing

Configuration Functions

Functions to set various parameters include the following:

- `sqlite3BtreeSetCacheSize`: Controls the page cache size as well as the synchronous writes (as defined in the `synchronous` pragma).
- `sqlite3BtreeSetSafetyLevel`: Changes the way data is synced to disk in order to increase or decrease how well the database resists damage because of OS crashes and power failures. Level 1 is the same as asynchronous (no `syncs()` occur, and there is a high probability of damage). This is the equivalent to pragma `synchronous=OFF`. Level 2 is the default. There is a very low but nonzero probability of damage. This is the equivalent to pragma `synchronous=NORMAL`. Level 3 reduces the probability of damage to near zero but with a write performance reduction. This is the equivalent to pragma `synchronous=FULL`.
- `sqlite3BtreeSetPageSize`: Sets the database page size.
- `sqlite3BtreeGetPageSize`: Returns the database page size.
- `sqlite3BtreeSetAutoVacuum`: Sets the autovacuum property of the database.
- `sqlite3BtreeGetAutoVacuum`: Returns whether the database uses autovacuum.
- `sqlite3BtreeSetBusyHandler`: Sets the busy handler.

There are more functions, all of which are very well documented in `btree.h` and `btree.c`, but those listed here give you some idea of the parts of the API that are implemented in the B-tree layer, as well as what this layer can do in its own right.

Download from Wow! eBook <www.wowebook.com>

Manifest Typing, Storage Classes, and Affinity

You'll remember from Chapter 4 our discussion on storage classes in SQLite and how its underlying data typing (or domains) offers a rather more flexible approach to the notion of data types that you find in other systems or databases. It's worth exploring the mechanisms that underpin storage classes further, including SQLite's approach to manifest typing and type affinity.

Manifest Typing

SQLite uses manifest typing. If you do a little research, you will find that the term *manifest typing* is subject to multiple interpretations. In programming languages, manifest typing refers to how the type of a variable or value is defined and/or determined. There are two main interpretations:

- *Manifest typing means that a variable's type must be explicitly declared in the code.* By this definition, languages such as C/C++, Pascal, and Java would be said to use manifest typing. Dynamically typed languages such as Perl, Python, and Ruby, on the other hand, are the direct opposite because they do not require that a variable's type be declared.
- *Manifest typing means that variables don't have types at all. Rather, only values have types.* This seems to be in line with dynamically typed languages. Basically, a variable can hold any value, and the type of that variable at any point in time is determined by its value at that moment. Thus, if you set variable `x`=1, then `x` at that moment is of type `integer`. If you then set `x='JujyFruit'`, it is then of type `text`. That is, if it looks like an `integer` and it acts like an `integer`, it is an `integer`.

For the sake of brevity, we will refer to the first interpretation as MT 1 and the second as MT 2. At first glance, it may not be readily apparent as to which interpretation best fits SQLite. For example, consider the following table:

```
create table foo( x integer,
                  y text,
                  z real );
```

Say we now insert a record into this table as follows:

```
insert into foo values ('1', '1', '1');
```

When SQLite creates the record, what type is stored internally for `x`, `y`, and `z`? The answer is `integer`, `text`, and `real`. Then it seems that SQLite uses MT 1: variables have declared types. But wait a second; column types in SQLite are optional, so we could have just as easily defined `foo` as follows:

```
create table foo(x, y, z);
```

Now let's do the same `insert`:

```
insert into foo values ('1', '1', '1');
```

What type are `x`, `y`, and `z` now? The answer: `text`, `text`, and `text`. Well, maybe SQLite is just setting columns to `text` by default. If you think that, then consider the following `insert` statement on the same table:

```
INSERT INTO FOO VALUES (1, 1.0, x'10');
```

What are `x`, `y`, and `z` in this row? `integer`, `real`, and `blob`. This looks like MT 2, where the value itself determines its type.

So, which one is it? The short answer is neither and both. The long answer is a little more involved. With respect to MT 1, SQLite lets you declare columns to have types if you want. This looks and feels like what other databases do. But you don't *have to*, thereby violating the MT1 interpretation as well. This is because in all situations SQLite can take any value and infer a type from it. It doesn't need the type declared in the column to help it out. With respect to MT 2, SQLite allows the type of the value to "influence" (maybe not completely determine) the type that gets stored in the column. But you can still declare the column with a type, and that type will exert some influence, thereby violating this interpretation as well—that types come from values only. What we really have here is the MT 3—the SQLite interpretation. It borrows from both MT 1 and MT 2.

But interestingly enough, manifest typing does not address the whole issue with respect to types. It seems to be concerned with only declaring and resolving types. What about type checking? That is, if you declare a column to be type integer, what exactly does that mean?

First let's consider what most other relational databases do. They enforce *strict type checking* as a standard part of standard domain integrity. First you declare a column's type. Then only values of that type can go in it. End of story. You can use additional domain constraints if you want, but under no conditions can you ever insert values of other types. Consider the following example with Oracle:

```
SQL> create table domain(x int, y varchar(2));
Table created.

SQL> insert into domain values ('pi', 3.14);
insert into domain values ('pi', 3.14)
                           *
ERROR at line 1:
ORA-01722: invalid number
```

The value `'pi'` is not an integer value. And column `x` was declared to be of type `int`. We don't even get to hear about the error in column `y` because the whole `insert` is aborted because of the integrity violation on `x`. When I try this in SQLite, we said one thing and did another, and SQLite didn't stop me:

```
sqlite> create table domain (x int, y varchar(2));
sqlite> INSERT INTO DOMAIN VALUES ('pi', 3.14);
sqlite> select * from domain;

x     y
----  -----
pi    3.14
```

SQLite's domain integrity does not include strict type checking. So, what is going on? Does a column's declared type count for anything? Yes. Then how is it used? It is all done with something called *type affinity*.

In short, SQLite's manifest typing states that columns can have types and that types can be inferred from values. Type affinity addresses how these two things relate to one another. Type affinity is a delicate balancing act that sits between strict typing and dynamic typing.

Type Affinity

In SQLite, columns don't have types or domains. Although a column can have a declared type, internally it has only a type affinity. Declared type and type affinity are two different things. Type affinity determines the storage class SQLite uses to store values within a column. The actual storage class a column uses to store a given value is a function of both the value's storage class and the column's affinity. Before getting into how this is done, however, let's first talk about how a column gets its affinity.

Column Types and Affinities

To begin with, every column has an affinity. There are four different kinds: `numeric`, `integer`, `text`, and `none`. A column's affinity is determined directly from its declared type (or lack thereof). Therefore, when you declare a column in a table, the type you choose to declare it will ultimately determine that column's affinity. SQLite assigns a column's affinity according to the following rules:

- By default, a column's default affinity is `numeric`. That is, if a column is not `integer`, `text`, or `none`, then it is automatically assigned `numeric` affinity.
- If a column's declared type contains the string `'int'` (in uppercase or lowercase), then the column is assigned `integer` affinity.
- If a column's declared type contains any of the strings `'char'`, `'clob'`, or `'text'` (in uppercase or lowercase), then that column is assigned `text` affinity. Notice that `'varchar'` contains the string `'char'` and thus will confer `text` affinity.
- If a column's declared type contains the string `'blob'` (in uppercase or lowercase), *or if it has no declared type,* then it is assigned `none` affinity.

■ **Note** Pay attention to defaults. For instance, `floatingpoint` has affinity `integer`. If you don't declare a column's type, then its affinity will be `none`, in which case all values will be stored using their given storage class (or inferred from their representation). If you are not sure what you want to put in a column or want to leave it open to change, this is the best affinity to use. However, be careful of the scenario where you declare a type that does not match any of the rules for `none`, `text`, or `integer`. Although you might intuitively think the default should be `none`, it is actually `numeric`. For example, if you declare a column of type `JUJYFRUIT`, it will not have affinity `none` just because SQLite doesn't recognize it. Rather, it will have affinity `numeric`. (Interestingly, the scenario also happens when you declare a column's type to be `numeric` for the same reason.) Rather than using an unrecognized type that ends up as `numeric`, you may prefer to leave the column's declared type out altogether, which will ensure it has affinity `none`.

Affinities and Storage

Each affinity influences how values are stored in its associated column. The rules governing storage are as follows:

- A `numeric` column may contain all five storage classes. A `numeric` column has a bias toward numeric storage classes (`integer` and `real`). When a `text` value is inserted into a `numeric` column, it will attempt to convert it to an `integer` storage class. If this fails, it will attempt to convert it to a `real` storage class. Failing that, it stores the value using the `text` storage class.
- An `integer` column tries to be as much like a `numeric` column as it can. An `integer` column will store a `real` value as `real`. *However,* if a `real` value has *no fractional component,* then it will be stored using an `integer` storage class. `integer` column will try to store a `text` value as `real` if possible. If that fails, they try to store it as `integer`. Failing that, `text` values are stored as `TEXT`.
- A `text` column will convert all `integer` or `real` values to `text`.
- A `none` column does not attempt to convert any values. All values are stored using their given storage class.
- No column will ever try to convert `null` or `blob` values—regardless of affinity. `null` and `blob` values are always stored as is in every column.

These rules may initially appear somewhat complex, but their overall design goal is simple: to make it possible for SQLite to mimic other relational databases if you need it to do so. That is, if you treat columns like a traditional database, type affinity rules will store values in the way you expect. If you declare a column of type `integer` and put integers into it, they will be stored as `integer`. If you declare a column to be of type `text`, `char`, or `varchar` and put integers into it, they will be stored as `text`. However, if you don't follow these conventions, SQLite will still find a way to store the value.

Affinities in Action

Let's look at a few examples to get the hang of how affinity works. Consider the following:

```
sqlite> create table domain(i int, n numeric, t text, b blob);
sqlite> insert into domain values (3.142,3.142,3.142,3.142);
sqlite> insert into domain values ('3.142','3.142','3.142','3.142');
sqlite> insert into domain values (3142,3142,3142,3142);
sqlite> insert into domain values (x'3142',x'3142',x'3142',x'3142');
sqlite> insert into domain values (null,null,null,null);
sqlite> select rowid,typeof(i),typeof(n),typeof(t),typeof(b) from domain;

rowid       typeof(i)   typeof(n)   typeof(t)   typeof(b)
----------  ----------  ----------  ----------  ----------
1           real        real        text        real
2           real        real        text        text
3           integer     integer     text        integer
4           blob        blob        blob        blob
5           null        null        null        null
```

The first `insert` inserts a `real` value. You can see this both by the format in the `insert` statement and by the resulting type shown in the `typeof(b)` column returned in the `select` statement. Remember that `blob` columns have storage class `none`, which does not attempt to convert the storage class of the input value, so column `b` uses the same storage class that was defined in the `insert` statement. Column `i` keeps the `numeric` storage class, because it tries to be `numeric` when it can. Column `n` doesn't have to convert anything. Column `t` converts it to `text`. Column `b` stores it exactly as given in the context. In each subsequent `insert`, you can see how the conversion rules are applied in each varying case.

The following SQL illustrates storage class sort order and interclass comparison (which are governed by the same set of rules):

```
sqlite> select rowid, b, typeof(b) from domain order by b;

rowid  b       typeof(b)
-----  ------  ---------
5      NULL    null
1      3.142   real
3      3142    integer
2      3.142   text
4      1B      blob
```

Here, you see that `null`s sort first, followed by `integer`s and `real`s, followed by `text`s and then `blob`s. The following SQL shows how these values compare with the integer 1,000. The `integer` and `real` values in `b` are less than 1,000 because they are numerically compared, while `text` and `blob` are greater than 1,000 because they are in a higher storage class.

```
sqlite> select rowid, b, typeof(b), b<1000 from domain order by b;

rowid  b      typeof(b)  b<1000
-----  -----  --------  ----------
5      NULL   null      NULL
1      3.142  real      1
3      3142   integer   1
2      3.142  text      0
4      1B     blob      0
```

The primary difference between type affinity and strict typing is that type affinity will never issue a constraint violation for incompatible data types. SQLite will always find a data type to put any value into any column. The only question is what type it will use to do so. The only role of a column's declared type in SQLite is simply to determine its affinity. Ultimately, it is the column's affinity that has any bearing on how values are stored inside of it. However, SQLite does provide facilities for ensuring that a column may only accept a given type or range of types. You do this using `check` constraints, explained in the sidebar "Makeshift Strict Typing" later in this section.

Storage Classes and Type Conversions

Another thing to note about storage classes is that they can sometimes influence how values are compared as well. Specifically, SQLite will sometimes convert values between numeric storage classes (`integer` and `real`) and `text` before comparing them. For binary comparisons, it uses the following rules:

- When a column value is compared to the result of an expression, the affinity of the column is applied to the result of the expression before the comparison takes place.
- When two column values are compared, if one column has `integer` or `numeric` affinity and the other doesn't, then `numeric` affinity is applied to `text` values in the non-`numeric` column.
- When two expressions are compared, SQLite does not make any conversions. The results are compared as is. If the expressions are of like storage classes, then the comparison function associated with that storage class is used to compare values. Otherwise, they are compared on the basis of their storage class.

Note that the term *expression* here refers to any scalar expression or literal *other than a column value*. To illustrate the first rule, consider the following:

```
sqlite> select rowid,b,typeof(i),i>'2.9' from domain order by b;

rowid  b      typeof(i  i>'2.9'
-----  -----  --------  ------------
5      NULL   null      NULL
1      3.142  real      1
3      3142   integer   1
2      3.142  real      1
4      1B     blob      1
```

The expression `'2.9'`, while being `text`, is converted to `integer` before the comparison. So, the column interprets the value in light of what it is. What if `'2.is` a non-numeric string? Then SQLite falls back to comparing storage class, in which `integer` and `numeric` types are always less than `text`:

```
sqlite> select rowid,b,typeof(i),i>'text' from domain order by b;

rowid  b      typeof(i  i>'text'
-----  -----  --------  ------------
5      NULL   null      NULL
1      3.14   real      0
3      314    integer   0
2      3.14   real      0
4      1B     blob      1
```

The second rule simply states that when comparing a numeric and non-numeric column, where possible SQLite will try to convert the non-numeric column to numeric format:

```
sqlite> create table rule2(a int, b text);
sqlite> insert into rule2 values(2,'1');
sqlite> insert into rule2 values(2,'text');
sqlite> select a, typeof(a),b,typeof(b), a>b from rule2;

a           typeof(a)   b           typeof(b)   a>b
----------  ----------  ----------  ----------  ----------
2           integer     1           text        1
2           integer     text        text        0
```

Column `a` is an `integer`, and `b` is `text`. When evaluating the expression `a>b`, SQLite tries to coerce `b` to `integer` where it can. In the first row, `b` is `'1'`, which can be coerced to `integer`. SQLite makes the conversion and compares integers. In the second row, `b` is `'text'` and can't be converted. SQLite then compares storage classes `integer` and `text`.

The third rule just reiterates that storage classes established by context are compared at face value. If what looks like a `text` type is compared with what looks like an `integer` type, then `text` is greater.

Additionally, you can manually convert the storage type of a column or an expression using the `cast()` function. Consider the following example:

```
sqlite> select typeof(3.14), typeof(cast(3.14 as text));

typeof(3.14)  typeof(cast(3.14 as text))
------------  --------------------------
real          text
```

MAKESHIFT STRICT TYPING

If you need something stronger than type affinity for domain integrity, then `CHECK` constraints can help. You can implement pseudo–strict typing directly using a single built-in function and a `CHECK` constraint. As mentioned earlier, SQLite has a function that returns the inferred storage class of a value—`typeof()`. You can use `typeof()` in any relational expression to test for a value type. Here's an example:

```
sqlite> select typeof(3.14) = 'text';
0
sqlite> select typeof(3.14) = 'integer';
0
sqlite> select typeof(3.14) = 'real';
1
sqlite> select typeof(3) = 'integer';
1
sqlite> select typeof('3') = 'text';
1
```

Therefore, you can use this function to implement a `CHECK` constraint that limits the acceptable types allowed in a column:

```
sqlite> create table domain (x integer check(typeof(x)='integer'));
sqlite> insert into domain values('1');
SQL error: constraint failed

sqlite> insert into domain values(1.1);
SQL error: constraint failed

sqlite> insert into domain values(1);
sqlite> select x, typeof(x) from domain;
```

```
x   typeof(x)
--  ----------
1   integer
sqlite> update domain set x=1.1;
SQL error: constraint failed
```

The only catch here is that you are limited to checking for SQLite's native storage classes (or what can be implemented using other built-in SQL functions). However, if you are a programmer and either use a language extension that supports SQLite's user-defined functions (for example, Perl, Python, or Ruby) or use the SQLite C API directly, you can implement even more elaborate functions for type checking, which can be called from within CHECK constraintsithin CHECK constraints.

Write Ahead Logging

With the release of SQLite 3.7.0, a new optional model for managing atomic transactions was introduced: the Write Ahead Log. The Write Ahead Log (WAL) is not new to the wider database technology world, but it's inclusion in SQLite is amazing for two reasons. First, this further enhances the sophistication and robustness of SQLite and the databases you use. Second, it's astounding that such a great feature has been included in SQLite and yet the code and resulting binaries are still so compact and concise.

How WAL Works

In the traditional mode of operation, SQLite uses the rollback journal to capture the prechange data from your SQL statements and then makes changes to the database file (we'll avoid discussing the various caching and file system layers here, because they apply regardless of model). When the WAL is used, the arrangement is reversed. Instead of writing the original, prechanged data to the rollback journal, using WAL leaves the original data untouched in the database file. The WAL file is used to record the changes to the data that occur for a given transaction. A commit action changes to being a special record written to the WAL to indicate the preceding changes are in fact complete and to be honored from an ACID perspective.

This change in roles between database file and log file immediately alters the performance dynamics of transactions. Instead of contending over the same pages in the database file, multiple transactions can simultaneously record their data changes in the WAL and can continue reading the unaltered data from the database file.

Checkpoints

The first question that most people think of when confronted with WAL technologies is usually "But when does the data ultimately get written *to the database*?" We're glad you asked. Obviously, a never-ending stream of changes being written to a constantly growing WAL file wouldn't scale or withstand the vagaries of file system failures. The WAL uses a checkpoint function to write changes back to the database. This process happens automatically, so the developer need not concern themselves with managing the checkpointing and write-back to the database. By default, the WAL invokes a checkpoint when the WAL file hits 1,000 pages of changes. This can be modified to suit different operating scenarios.

Concurrency

Because changes now write to the WAL file, rather than the database file, you might be wondering how this affects SQLite concurrency. The good news is the answer "overwhelmingly for the better!"

New readers accessing data from the WAL-enabled database initiate a lookup in the WAL file to determine the last commit record. This point becomes their end marker for read-only consistency, such that they will not consider new writes beyond this point in the file. This is how the read becomes a consistent read—it simply takes no notice of anything after the commit considered its end marker. Each reader assess is independent, so it's possible to have multiple threads each with a different notion of their own individual end mark within the WAL file.

To access a page of data, the reader uses a wal-index structure to scan the WAL file to find out whether the page exists with changes and uses the latest version if found. If the page isn't present in the WAL file, it means it hasn't been altered since the last checkpoint, and the reader accesses the page from the database file.

The wal-index structure is implemented in shared memory, meaning that all threads and processes must have access to the same memory space. That is, your programs must all execute on the same machine in order to access a WAL-enabled database or, more accurately, to use the wal-index. This is the feature that rules out operation across network file systems like NFS.

As for writers, thanks to the sequential nature of the WAL file, they simply append changes to the end. Note, however, that this means they must take it in turn appending their changes, so writers can still block writers.

Activation and Configuration WAL

SQLite will still default to the rollback journal method until you decide to switch your database to use the WAL. Unlike other `PRAGMA` settings, turning on the WAL is a persistent database-level change. This means you can turn on WAL in one program or from the SQLite command shell, and all other programs will then be using the database in WAL mode. This is very handy because it means you don't have to alter your applications to turn on WAL.

The setting `PRAGMA journal_mode=WAL` is the command necessary to activate WAL. A call to this pragma will return as a string the final journal mode for the database.

```
SQLite version 3.7.3
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> PRAGMA journal_mode=WAL;
wal
```

If the change to WAL is successful, the string `wal` is returned, as shown. If for some reason this fails, such as the underlying host not supporting the necessary shared memory requirements, the journal mode will be unchanged. For instance, on a default database, this means the command will return the string `delete`, meaning the rollback journal is still active.

As well as the `journal_mode PRAGMA`, several API options and related `PRAGMAs` are supported to control WAL and checkpoint behavior.

- `sqlite3_wal_checkpoint()`: Forces a checkpoint (also available as `wal_checkpoint PRAGMA`)
- `sqlite3_wal_autocheckpoint()`: Alters autocheckpoint page threshold (`wal_autocheckpoint PRAGMA`)
- `sqlite3_wal_hook()`: Registers a callback to be invoked when a program commits to the WAL.

With these options, it's possible to fine-tune WAL and checkpoint behavior to suit even the most unusual of applications.

WAL Advantages and Disadvantages

Naturally, a change such as WAL brings with it its own advantages and disadvantages. We think that the advantages are significant, but it's best to be aware of impact so you can form your own judgment.

Moving to WAL has the following advantages:

- Readers do not block writers, and writers do not block readers. This is the "gold standard" in concurrency management.
- WAL is flat-out faster in most operating scenarios, when compared to rollback journaling.
- Disk I/O becomes more predictable and induces fewer `fsync()` system calls. Because all WAL writes are to a linearly written log file, much of the I/O becomes sequential and can be planned accordingly.

To balance the equation, here are some of the notable disadvantages of WAL:

- All processing is bound to a single host. That is, you cannot use WAL over a networked file system like NFS.
- Using WAL introduces two additional semipersistent files, `<yourdb>-wal` and `<yourdb>-shm`, for the WAL and related shared memory requirements. This can be unappealing for those using SQLite databases as an application file format. This also affects read-only environments, because the `-shm` file must be writable, and/or the directory in which the database exists.
- WAL performance will degrade for very large (approaching gigabyte) transactions. Although WAL is a high-performance option, very large or very long-running transactions introduce additional overhead.

A number of other edge cases are worth investigating if you are thinking of using SQLite with WAL-enabled databases in unusual environments. You can review more details about these at `www.sqlite.org/wal.html`.

Download from Wow! eBook <www.wowebook.com>

Operational Issues with WAL-Enabled SQLite Databases

There are a few operational considerations when working with WAL-enabled SQLite databases. These are primarily performance and recovery related.

Performance

We noted earlier that enabling WAL usually translates to better performance. Under our disadvantages, we outlined that for very large changes in data, or very long-running transactions, a slight drop in performance might result. Here's the background on how this can happen and what to do.

With WAL enabled, you implicitly gain a performance advantage over rollback journaling because only one write is required to commit a transaction, instead of two (one to the database and rollback journal). However, until a checkpoint occurs that frees space in the WAL file, the file itself must keep growing as more and more changes are recorded. Normally this isn't an issue: someone will be the lucky committer who pushes WAL through the 1,000-page threshold for a checkpoint, and the WAL will clear, and the cycle will continue. But the checkpoint completes its work if it finds an end-point marker for any currently active reader. That means a very long-running transaction or read activity can keep the checkpoint from making progress, even on subsequent attempts. This in turn means the WAL file keeps growing in size, with commensurately longer seek times for new readers to find pages.

Another performance consideration is how much work is triggered when a transaction hits the checkpoint threshold. You may find that the variable nature of lots of very fast transactions and then one slightly longer one that must perform the checkpoint work for its predecessors unattractive in some scenarios.

For both of these performance hypotheticals, you are essentially weighing up read and write performance. Our recommended approach is to first remember that you can't have the best of both, so some compromise will be needed, and then if necessary use the `sqlite3_wal_checkpoint()`and `sqlite3_wal_autocheckpoint()` tools to balance the load and find a happy equilibrium.

Recovery

Versions of SQLite prior to 3.7.0 have no knowledge of WAL, nor the methods used for recovering WAL-based databases. To prevent older versions trampling all over a WAL-enabled SQLite database during crash recovery, the database file format number was bumped up from 1 to 2. When an older version of SQLite attempts to recover a database with this change, it realizes it is not a "valid" SQLite database and will report an error similar to the following.

```
file is encrypted or is not a database
```

Don't panic! Your database is fine, but the version of SQLite will need to be upgraded in order to work with it. Alternatively, you can change back to the rollback journal.

```
PRAGMA journal_mode=DELETE;
```

This resets the database file format number to 1.

Summary

That concludes our journey through SQLite, not only in this chapter but the book as well. We hope you have enjoyed it. As you saw in Chapter 1, SQLite is more than merely a free database. It is a well-written software library with a wide range of applications. It is a database, a utility, and a helpful programming tool.

What you have seen in this chapter barely scratches the surface of the internals, but it should give you a better idea about how things work nonetheless. And it also gives you an appreciation for how elegantly SQLite approaches a very complex problem. You know firsthand how big SQL is, and you've seen the complexity of the models behind it. Yet SQLite is a small library and manages to put many of these concepts to work in a small amount of code and excels!